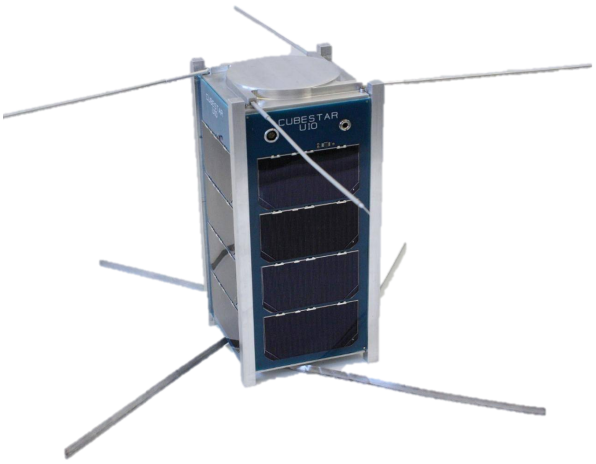
UiO : Department of Physics
University of Oslo

Design and Implementation of the Electrical Power System for the CubeSTAR Satellite

Master Thesis

Knut Olav
Skyttemyr

March 2013



CUBESTAR

UNIVERSITAS OSLOENSIS • MDCCCXI •

Abstract

This thesis describes the design and implementation of an electronic power system for the CubeSTAR satellite. The main task of the power system is to supply continuous power to the satellite in orbit. The power system consists of several parts: Solar cells, battery chargers, battery pack, power distribution bus, sensors monitoring different parts of the system and a microcontroller used to control the system.

The solar cells are used to generate power to the satellite and charge the battery pack. The battery charging system is based on the SPV1040 charger with embedded maximum power point tracking. A charger efficiency of 90% has been achieved. The battery pack is made of six lithium iron phosphate battery cells connected in parallel. It has been decided to distribute unregulated power. This leaves each subsystem responsible for regulating the power with respect to their requirements. From the estimated power budget, the total average power consumption of the satellite should be below 2 W. The electronic power system itself uses an average power of 66 mW.

A microcontroller is used to control the power system and it is a slave unit receiving commands from the satellite main processor. The microcontroller gathers all sensor data and prepares the data for transmission to the ground station. There are various voltage, current and temperature sensors implemented for monitoring the power system.

Acknowledgements

I would like to thank my supervisor Associate Professor Torfinn Lindem. Thank you for the opportunity to work with the CubeSTAR project. It has been very interesting and I have learned a lot from working on my thesis. Thank you for trusting me by giving me the responsibility of the electronic power system. I have been given freedom to find solutions on my own and I have been given guidance when needed.

I would like to thank Senior Engineer Stein Lyng Nielsen at the Electronics Laboratory. You have been a valuable resource when it comes to electronics and printed circuit boards. Thank you for the cooperation of making the CubeSTAR back panel and thank you for being patient about all the design changes I have made. Stein Lyng Nielsen has been responsible for the printed circuit board layout of the back panel. You carry out your work with high accuracy and quality. Head Engineer Halvor Strøm has also been very helpful with electronic component procurement and printed circuit board assembly.

PhD candidate Johan Ludvig Tresvig has been very helpful with reviewing my designs. It has been a great comfort having you around in the office and thank you for always having time to discuss all the small details in my designs. I would like to thank master student Joakim Myrland. I appreciate all help and interesting discussions regarding microcontroller programming.

Finally, I would like to thank my beautiful wife Linda. You have been patient and encouraging throughout this period of writing my master thesis.

Oslo, March 2013
Knut Olav Skyttemyr

Contents

List of Figures

List of Tables

Nomenclature

ADC	Analog-to-Digital Converter
ADCS	Attitude Determination & Control System
CC-CV	Constant Current Constant Voltage
COMM	Communication system
COTS	Commercial Off The Shelf
EPS	Electronic Power System
I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
LEO	Low-Earth Orbit
$LiFePO_4$	Lithium Iron Phosphate
m-NLP	multi-Needle Langmuir Probe
MCU	Microcontroller
MPPT	Maximum Power Point Tracking
OAP	Orbit Average Power
OBC	On Board Controller
P-POD	Poly Picosatellite Orbital Deployer
PCB	Printed Circuit Board
PV	Photo Voltaic
RBF	Remove Before Flight
RTC	Real-Time Counter
SEL	Single Event Latchup

SEU	Single Event Upset
SMPS	Switched-Mode Power Supply
SoC	State of Charge
TCS	Thermal Control System
TMR	Triple Modular Redundancy
UTJ	Ultra Triple Junction

Chapter 1

Introduction

1.1 CubeSTAR

CubeSTAR is a satellite built by students at the Department of Physics, University of Oslo (UiO). The project started in 2008 and the satellite launch is planned to take place in 2014. The satellite is designed to comply with the cubesat standard from California Polytechnic State University and Standford University [5]. CubeSTAR is a 2 unit cubesat, meaning the size is $10\,\text{cm} \times 10\,\text{cm} \times 20\,\text{cm}$. The CubeSTAR is modular, meaning the satellite consists of a back panel with several connectors for connecting different subsystems. The satellite is divided into the following subsystems: The scientific payload, the communication system (COMM), the Attitude Determination & Control System (ADCS), the On Board Controller (OBC) and the Electronic Power System (EPS).

The scientific payload on CubeSTAR aims to measure effects of space weather in the ionospheric plasma. More specifically, the payload measures the electron density using the multi-Needle Langmuir Probe (m-NLP) system developed at UiO. PhD student Tore André Bekkeng is responsible for the realization of this system [3]. Solar storms are causing electron clouds to form in the ionospheric plasma. These electron clouds disturb GPS precision and other satellite communications. This problem is particularly significant around the polar areas where the magnetic fields of the Earth converge. The measurement data from the m-NLP can help us in getting a better understanding of the phenomenon. It can help finding a way to compensate for these disturbances, thus providing better GPS precision during varying space weather.

The COMM consists of a half-duplex transceiver operating in the UHF amateur satellite band. The COMM is used to communicate with the ground station by transmitting payload and housekeeping data on the downlink and receiving commands on the uplink. The ADCS is responsible for detumbling and controlling the satellite attitude (orientation). The attitude must be controlled because the m-NLP system needs to point in the travelling direction to avoid picking up electron turbulence created by the satellite. The OBC is the satellite main processor. The OBC is responsible for supervising the subsystems, collecting housekeeping data and maintaining satellite operation.

Martin Oredsson finished his thesis "Electrical Power System for the CubeSTAR Nanosatellite" in September 2010 [15]. Oredsson successfully managed to create a battery charging system that utilize a Maximum Power Point Tracking (MPPT) algorithm to harvest power from solar cells. Oredsson tested Lithium Iron Phosphate ($LiFePO_4$) battery technology and he states that $LiFePO_4$ battery cells should be suitable for an unregulated power supply system. This because of the flat voltage discharge curve compared to other battery cell technologies. The cell voltage drops quickly when exposed to low temperatures and Oredsson recommends considering a battery heater. Oredsson never states the true efficiency of his charging system. His system is relatively complex, so some of the extra power gained by using the MPPT algorithm is lost by powering the charging system itself. The EPS made by Oredsson demonstrates interesting technology but is not ready to be launched into space. It has to be investigated whether a new EPS can be made using the same technology but with higher efficiency and less complexity.

1.2 Goals of this Thesis

This thesis describes the design and implementation of a new EPS. The purpose of the EPS is to provide continuous power to the satellite during orbit and the primary goal of this thesis is to build a reliable EPS for the CubeSTAR.

In the world of Integrated Circuits (IC) new exciting products enters the marked every year. The SPV1040 battery charger was introduced by ST Microelectronics a few years ago [24]. This is a high efficiency solar battery charger with an embedded MPPT algorithm. This charger looks promising and this eight pin IC could replace the charging system made by Oredsson. It has to be decided if $LiFePO_4$ batteries are going to be used. Thermal challenges need to be solved and the battery pack itself has to be realized. The power distribution structure and communication bus structure have to be settled. The EPS need different sensors to monitor the state of the battery pack, chargers and power distribution system.

When the CubeSTAR project started, it was decided that the Electronics Laboratory at UiO is responsible for the Printed Circuit Board (PCB) layout of the back panel. This thesis will describe the components needed, how to put the circuit schematic together and verify correct functionality of the back panel. When it comes to the battery pack, this thesis will include both the circuit schematics and the PCB layout.

To broaden the scope of this thesis some programming tasks have been added: The EPS firmware that includes sensor data housekeeping and power distribution system control. The EPS firmware will be implemented on a microcontroller (MCU) and will be a slave unit commanded by the satellite main processor, the OBC.

The CubeSTAR is designed using Commercial Off The Shelf (COTS) components. Both the hardware and the firmware should be designed with redundancy and fault tolerance in mind, but still keep the design as simple as possible to reduce the risk of errors.

Chapter 2

Satellite Power Systems

The satellite power system consists of Photo Voltaic (PV) cells, battery chargers and batteries. In this chapter we will look at the key technology needed to make a satellite power system. In addition, we will look at battery monitoring and space related challenges we have to deal with.

2.1 Solar Power

The PV-cell chosen for the CubeSTAR mission is the Ultra Triple Junction (UTJ) cell from Spectrolab [21]. This cell has an area of 26.62 cm^2, an efficiency of 28.3% and has been part of many successful space missions. Due to the triple junction the efficiency of this cell is among the highest available on the market.

Solar power is based on the photovoltaic effect and the PV-cells are made with semiconductor materials [4]. Energy in form of electromagnetic waves from the Sun is hitting the atoms in the PV-cell. If the energy absorbed in the atom is higher than the bandgap energy (E_g), an electron is moved from the valence band up to the conduction band (figure 2.1). Free electrons in the conduction band can form an electric current. Most of the energy from the Sun is transformed to heat or reflected on the surface of the PV-cell. The efficiency of the PV-cell tells us how many percent of the solar energy that is successfully transformed to electrical energy.

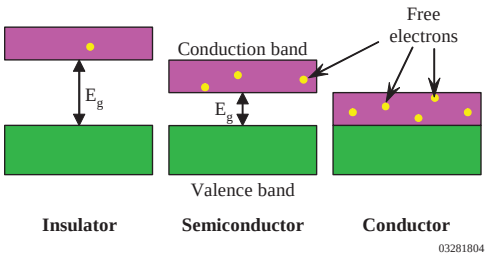


Figure 2.1: Difference in bandgaps between valence band and conduction band of isolator, semiconductor and conductor [4].

The electromagnetic waves from the Sun consist of a large range of wavelengths. The solar irradiance tells us about the energy per area. The irradiance spectrum is different in outer space than here at the Earth's surface. Air mass zero (AM0) is the irradiance outside of the atmosphere. On the Earth's surface the irradiance is lower due to the atmosphere. It is depending on the Sun's angle which varies with time. AM1.5 is suitable for the average solar conditions at the Earth's surface. If we integrate the irradiance spectrum we find the AM0 and AM1.5 solar constants which are 1366 W/m^2 and 1000 W/m^2 respectively. See figure 2.2 for a plot of the irradiance spectrum[1].

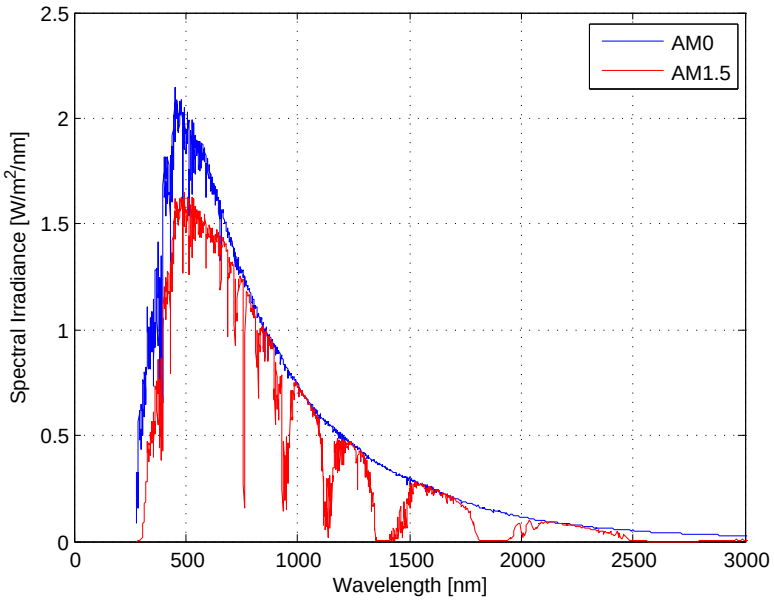


Figure 2.2: Solar irradiance spectrum. AM0 is the irradiance outside the atmosphere, AM1.5 is the average irradiance on the Earth's surface.

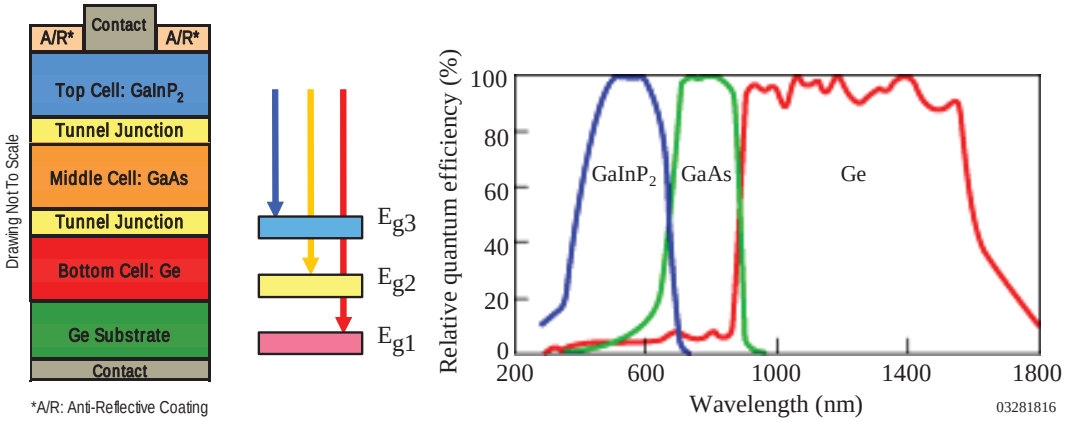
Different semiconductor materials have different bandgap energy levels. From the well-known formula for photon energy:

$$E = hf, \tag{2.1}$$

where h is Planck's constant and f is the frequency of the electromagnetic wave, we see that the energy is determined by the wavelength of the electromagnetic wave. To be able to utilize a larger range of irradiance spectrum, we use different semiconductor materials sandwiched into a multiple junction PV-cell. The bandgaps of the layers are decreasing with the top layer having the largest bandgap. The top layer is absorbing the highest energy photons while lower energy photons are transmitted down to the lower layers. The Spectrolab UTJ PV-cell consists of Germanium (Ge), Gallium Arsenide (GaAs) and

[1]Solar irradiance data: `http://rredc.nrel.gov/solar/spectra/am1.5/` Accessed: 05.12.2012

Gallium Indium Phosphide ($GaInP_2$). As figure 2.3 shows, we are now able to harvest energy from wavelengths around 300 nm to 1800 nm using the triple layer structure.



(2.3a) *(2.3b)* *(2.3c)*

Figure 2.3: a) Triple junction PV-cell stack-up [21]. b) The different layers have different bandgaps [4]. c) Efficiency of the different layers vs wavelength [4].

In figure 2.4 we see a PV-cell modelled as an electric circuit with a current source, diodes, leakage resistor (R_{sh}) and the internal resistance (R_s) [6]. In a high quality PV-

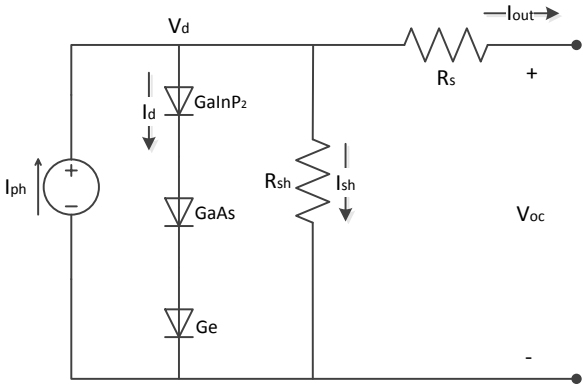


Figure 2.4: Simplified schematic of triple junction PV-cell.

cell the leakage resistance is large and the internal resistance low. The current from the current source (I_{ph}) is depending on the incoming solar irradiance. The diode current is found from the Shockley equation:

$$I_d = I_s(e^{V_d/(nV_t)} - 1), \tag{2.2}$$

where I_s is the reverse bias saturation current, the ideality factor $n = 1$ for an ideal diode, and the thermal voltage $V_t = kT/q$, where k is the Boltzmann constant, T is

temperature and q is elementary charge. The leakage current is found from:

$$I_{sh} = \frac{V_d}{R_{sh}}. \tag{2.3}$$

When the output terminals of the PV-cell is short-circuited, V_d is zero. Then we quickly see that both the diode current (I_d) and leakage current (I_{sh}) is zero. We can find the short circuited output current from:

$$\begin{aligned} I_{sc} &= I_{ph} - I_d - I_{sh} \\ &= I_{ph}. \end{aligned} \tag{2.4}$$

When the PV-cell is open circuited, the output current is zero and the open-circuit voltage (V_{oc}) is equal to V_d. V_{oc} can be found solving the Shockley equation for V_d:

$$\begin{aligned} V_{oc} &= V_d \\ &= nV_t \cdot ln\left(\frac{I_d}{I_s} + 1\right). \end{aligned} \tag{2.5}$$

Spectrolab UTJ PV-cells incorporates a bypass diode. This diode is used to conduct the current past a shaded or damaged PV-cell connected in series with other PV-cells. A shaded or damaged PV-cell will be reverse biased and introduce a high resistance to the current. As we can see in figure 2.5, the bypass diode of a reverse biased PV-cell will be forward biased, thus conducting the current past the shaded or damaged PV-cell.

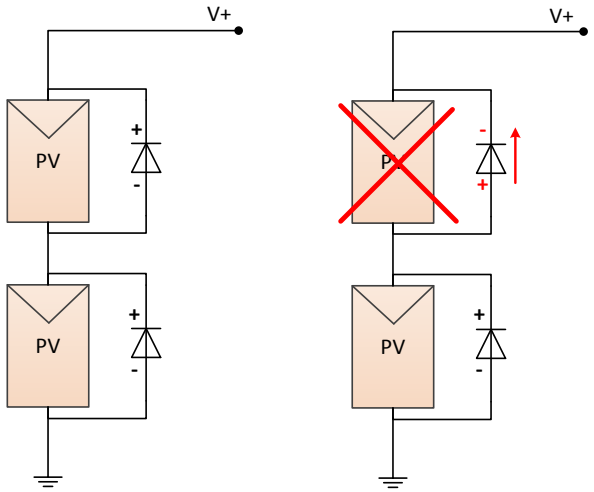


Figure 2.5: Bypass diode operation. The current is conducted through the diode past the shaded or damaged PV-cell.

2.2 Maximum Power Point Tracking

We know from basic electronics that power is voltage multiplied by current. The maximum voltage of a solar cell is the V_{oc}. As we see from figure 2.6, the current is rapidly decreasing towards zero when the voltage approach V_{oc}. The load resistance is infinite

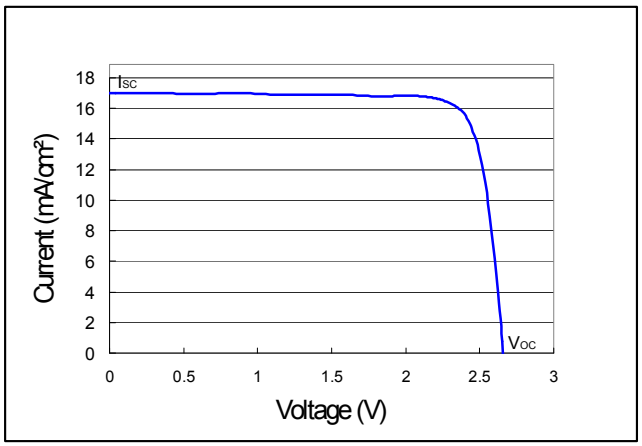


Figure 2.6: I-V curve of Spectrolab UTJ PV-cell at 28° C and AM0 conditions: V_{oc}=2.66 V and I_{sc}=17.05 mA/cm² [21].

when the solar cell is open-circuited, thus the current must be zero. The maximum power (P_{mp}) should be somewhere around the point where the current starts to decrease:

$$P_{mp} = V_{mp} I_{mp}. \tag{2.6}$$

V_{oc} is dependent on temperature. From the PV-cell datasheet, we find a V_{oc} of 2.66 V at 28°C [21]. We find a temperature coefficient of $-5.9\,\mathrm{mV/°C}$. This means that at 53°C:

$$\begin{aligned} V_{oc} &= 2.66\,\mathrm{V} - 5.9\,\mathrm{mV} \cdot 25\,^{\circ}\mathrm{C} \\ &= 2.51\,\mathrm{V}. \end{aligned}$$

At $3°C$:

$$\begin{aligned} V_{oc} &= 2.66\,\mathrm{V} + 5.9\,\mathrm{mV} \cdot 25\,^{\circ}\mathrm{C} \\ &= 2.81\,\mathrm{V}. \end{aligned}$$

In orbit, the PV-cells will dramatically change temperature when going from eclipse into direct sunlight. From figure 2.7 we see that another cubesat[2] experienced external temperatures from -25° to +25°C.

[2]Swisscube flight housekeeping data: `http://ctsgepc7.epfl.ch`. Accessed: 28.09.2012

Like V_{oc}, I_{sc} is depending on temperature but the main factor determining the I_{sc} is the incoming solar irradiance. Here on Earth clouds can lower the solar irradiance, but in orbit we have a constant solar irradiance. Both the V_{oc} and I_{sc} will decrease if the radiation angle of incidence increases. We cannot expect to have a fixed orientation towards the Sun, so the maximum power point will change with the angle of incidence. The changing temperature and angle of incidence tell us that a maximum power point tracking algorithm should be used to adjust the load dynamically.

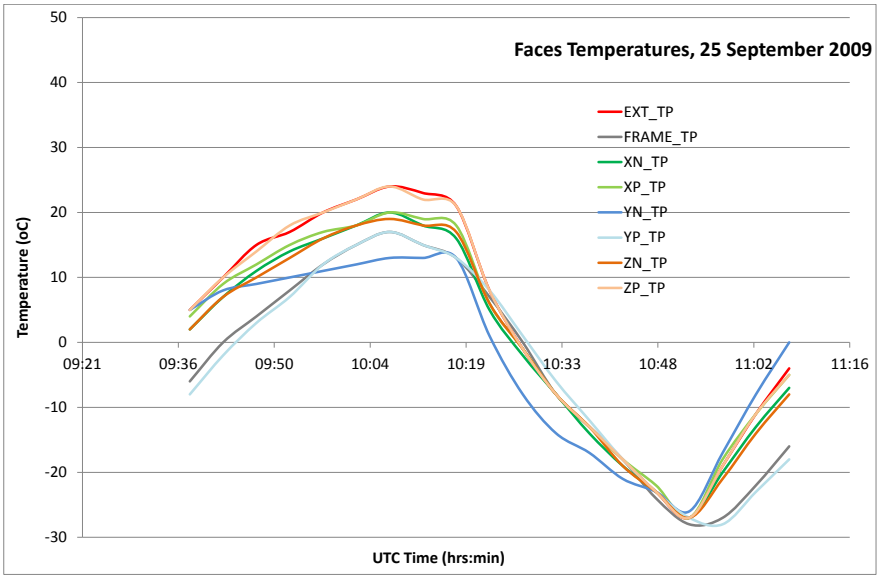


Figure 2.7: External temperatures on Swisscube in orbit.

Several methods of maximum power point tracking are available. The most common method, and the method used in the SPV1040 charger, is the Perturb and Observe method. The popularity of this method is due to low complexity and low power loss. The perturb and observe method works as follows: First we gather the power at time n and compare to the power at time $n-1$. If $P(n) \geq P(n-1)$, we increase the load to increase the power. If $P(n) < P(n-1)$ we decrease the load. By using this simple algorithm, we will toggle around the maximum power point at all times independent on temperature. The frequency of the algorithm decides how often the power point will be adjusted. A rather high frequency is preferred in order to quickly adjust to changes. See figure 2.8 for a graphical overview of the MPPT algorithm. The algorithm will quickly take us from point 1 to 5 and then toggle between point 4 and 5 until V_{oc} is changed.

One of the negative sides of this Perturb and Observe method is the constant oscillation around the maximum power point. As we see in figure 2.8, the power point will move back and forth between point 4 and 5. This will cause a slight power loss compared to the maximum power point.

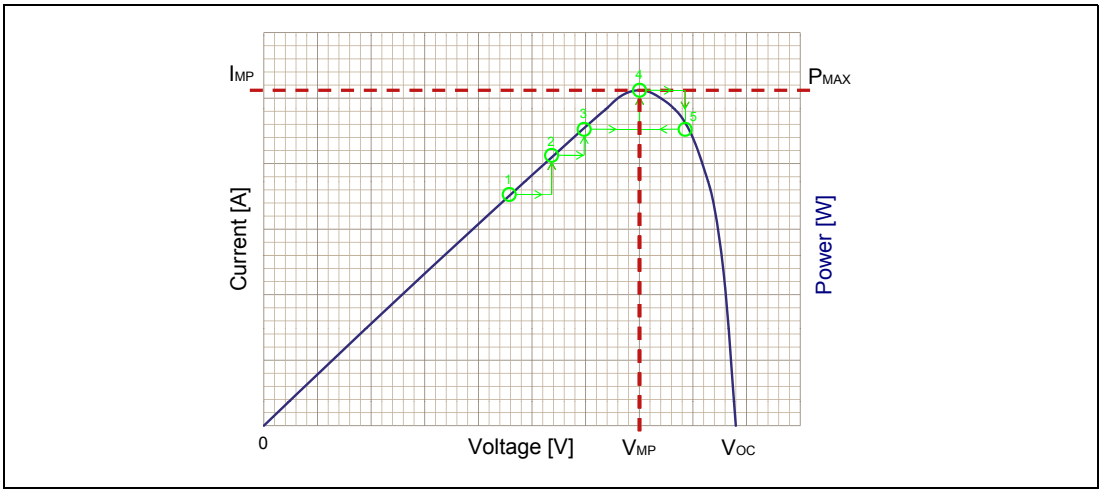


Figure 2.8: Maximum power point tracking, Perturb and Observe method. The algorithm adjusts the loading of the PV-cell to always stay around the maximum power point [23].

2.3 Battery Chemistry

The satellite will need some kind of energy storage because there will be no power available from the PV-cells during the eclipse. The most common way of storing energy is using batteries. The excess power from the PV-cells during the sunlit portion of the orbit will be stored in the batteries and used during the eclipse. There are several battery technologies available today. Lithium-ion is one of the most common. There are several different lithium-ion chemistries available. The battery proposed for the CubeSTAR satellite is a $LiFePO_4$ chemistry from the manufacturer A123 Systems [1]. The model proposed for CubeSTAR is the APR18650M1A, (3.3 V, 1.1 Ah). A123 Systems have a patented chemistry called Nanophosphate [25].

A battery is an electrochemical storage medium [12]. When we discharge a battery, chemical energy is converted to electrical energy and visa versa when we charge a battery. Each battery is built up by several voltaic cells. Each voltaic cell consists of two electrodes immersed in an electrolyte. The terminals of the battery are connected to the electrodes. The negative electrode is called the anode and the positive electrode is called the cathode. The electrical characteristics of the battery are depending on what materials the electrodes and the electrolyte are made of. The electrolyte permits transfer of ions between the two electrodes. A chemical reaction, called oxidation, takes place between the anode and the electrolyte. The result of the oxidation is free electrons and positively charged ions which migrate into the electrolyte. The free electrons can form a current in an electrical circuit connecting the battery terminals. The free electrons end up in the cathode and are part of another chemical process. The electrons are combined with positive ions in a process called reduction.

2.3.1 A123 Nanophosphate battery

There are many different materials used as electrodes and electrolyte. Today lithium is widely used as electrolyte. A123 Nanophosphate batteries offers some advantages compared to other chemistries. In traditional lithium-ion batteries, the reduction process of combining the lithium ions with electrons in the cathode is quite slow. This limits the power output of the battery. Nanophosphate batteries have increased the cathode surface area, thus allowing a faster reduction process and higher power output. This means the Nanophosphate batteries can give a lot higher continuous current. Compared to other batteries, the maximum current rate is not reduced notably when the battery is deeply discharged. The Nanophosphate batteries can tolerate much more abuse than conventional lithium-ion batteries. Nanophosphate batteries can withstand overcharge and high temperatures without permanent degradation or safety hazards. NASA has performed extensive stress testing of the A123 battery [11].

Other important features of the Nanophosphate batteries are the cycle life and storage capabilities. The cycle life of a rechargeable battery is the number of Depth of Discharge (DOD) cycles the battery can withstand before the total capacity of the battery is reduced below 80% of initial capacity. A123 claim their battery can withstand more than 7000 DOD cycles. Traditional lithium-ion batteries have a cycle life below 1000 DOD cycles. The storage capability of a battery tells us how much energy a battery loses during storage due to effects like self-discharge. Nanophosphate batteries have a low self-discharge rate and A123 promises a self-discharge of 3-4% in one year at 25°C [26].

The Nanophosphate battery has a relatively flat discharge curve compared to other batteries. In figure 2.9 we can see the discharge curve of APR18650M1A with 1.1 A constant current drawn from the battery and short current transients of 22 A (20 C). As we see at 25°C and 40°C, the battery voltage is fairly constant at 3.25 V. When the battery approaches 100% DOD, the battery voltage quickly drops towards 2.0 V. Another thing we can see is that when the temperature falls below 0°C, the battery voltage and total capacity decrease. Low temperatures is a problem for most battery technologies and has to be kept in mind when designing the EPS.

You seldom find technology without drawbacks. One negative side of the Nanophosphate batteries is the energy density. Compared to other battery chemistries the energy density is a bit lower, which means a heavier battery pack. This could demand more of the mechanical fixing of the battery pack. We have not been able to find flight records of this battery, but we know that other cubesat projects, like NUTS from the Norwegian University of Science and Technology, plan to use the same battery in their mission. The following list summarize why we choose A123 batteries for our mission:

- Low internal resistance, giving high power and low loss
- High safety, abuse tolerant and failure resistant
- Low self-discharge
- High DOD-cycle count

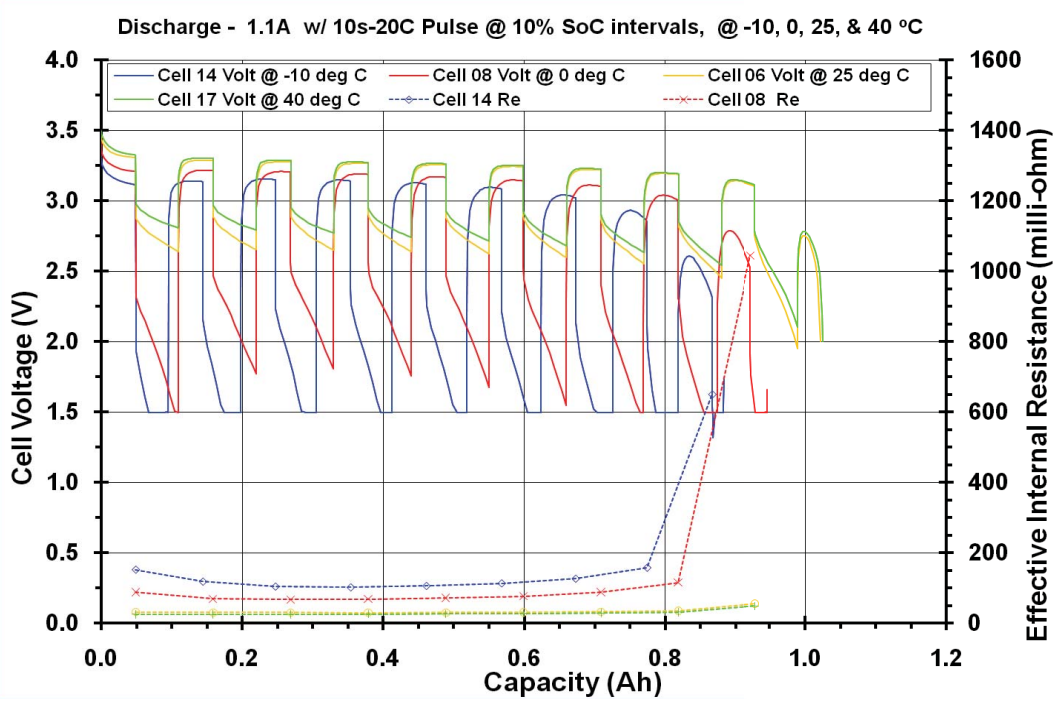


Figure 2.9: Discharge curve of A123 APR18650M1A battery cell. Test performed by NASA [11].

2.4 Charging Techniques

Different battery chemistries has led to different charging techniques [17]. The importance of charging technique depends on how much abuse the battery can withstand without damage. For instance over-charging can lead to damaged batteries or even fire. We have read that the A123 cell can tolerate much abuse [11]. But to achieve the best performance of the battery we should follow the recommendations from the manufacturer. From the battery datasheet we read that the battery should be charged using a Constant Current Constant Voltage (CC-CV) charging technique. See figure 2.10 for a graphical overview of the CC-CV charging technique.

The first phase is the constant current phase. The battery is charged using constant current until reaching the charging cut-off voltage. A123 recommends 1.5 A until reaching 3.6 V. After reaching the charge cut-off voltage the charger switch to the constant voltage phase. The reason for this two phase method is that we do not want to apply over-current or over-voltage to the battery. When the charger is using constant current, the voltage is adjusted to match the battery voltage. If we suddenly applied 3.6 V over a deeply discharged battery, i.e. < 2.0 V, the current into the battery would be very high due to the low internal resistance of the battery. High current could damage the battery. After reaching the charge cut-off voltage, over-current is no longer an issue. If we continue to charge with constant current, the battery voltage will continue to rise above the nominal voltage. If the voltage gets too high, the battery will be damaged. Therefore,

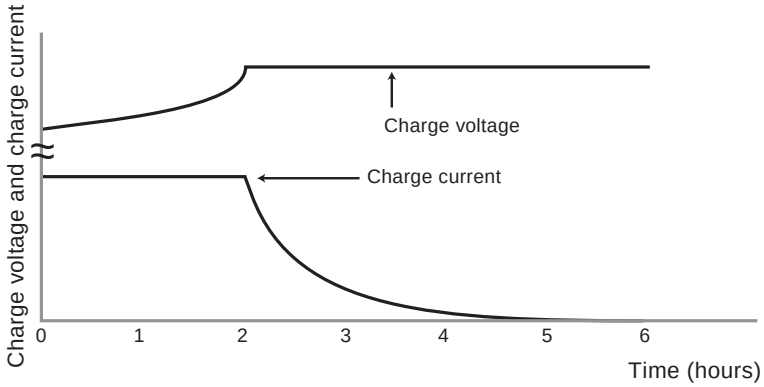


Figure 2.10: CC-CV battery charging technique. The current is kept constant until reaching the charge cut-off voltage. Then the current gradually decreases to keep a constant voltage [17].

the charger switches to constant voltage charging. The current will be gradually lower until the battery is fully charged.

2.5 Switched-Mode Power Supply

Switched-Mode Power Supplies (SMPS) are attractive due to their high efficiency [9]. The other common alternative is linear regulators. Generally speaking a linear regulator is less efficient than a SMPS. The efficiency of a linear regulator is depending on the difference between the input and output voltage and the load current. The linear regulator regulates the output by dissipating the input to output voltage difference as waste heat. Higher efficiency and lower heat generation is the two main arguments for using a SMPS. It is important to not waste power, especially in battery powered systems. In addition to high efficiency, the SMPS can be used to step up the voltage. Linear regulators only regulate to a lower voltage.

There are some negative sides with SMPS. Due to the high switching frequency, there are more problems with noise and Electro Magnetic Interference (EMI). The switching signal is a square wave and we know from mathematics that a square wave is built up by the fundamental frequency component and its harmonic components. A square wave will give high frequency noise in the system. This can both lead to problems internally and externally. Internally the problems can for instance be contamination of analogue signals. The problem can extend to the environments in form of noise transferred to other circuits nearby using electromagnetic coupling [16].

The two most common topologies are Buck and Boost. The Buck converter is regulating down to a lower voltage on the output, like a linear regulator. The boost converter is stepping the input voltage up to a higher voltage on the output compared to the input. You can have a combination, a Buck-Boost regulator, which is able to both step up and step down the voltage. There are several other topologies not mentioned here.

2.5.1 Boost Regulator

The SPV1040 charger is a boost regulator. To boost the voltage to a higher level than the input you need an energy storage element, often an inductor. Charge pumps, which are another type of SMPS, are using capacitors as energy storage. The regulation consists of two states, switch closed or switch open. First the switch is closed and the current flows through the inductor and magnetic energy is stored in the inductor. The diode is reverse-biased, so no current is allowed through the diode. The switching frequency is high, so the output capacitor is not discharged notably during the switch-closed state. See figure 2.11 for the switch-closed state.

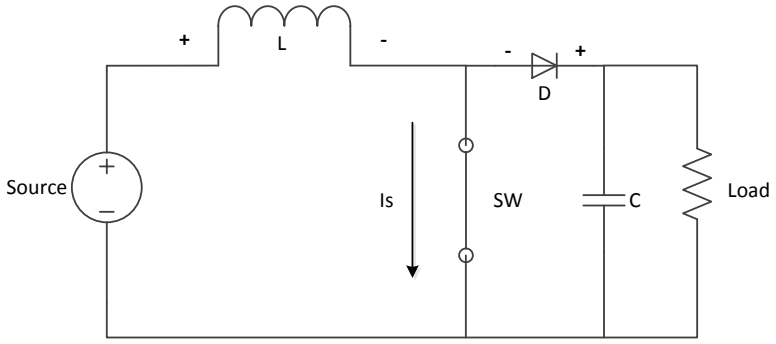


Figure 2.11: Boost regulator: Switch-closed state, store energy in the inductor (L).

The next state is switch-open. Inductors resist changes in current and when the switch opens, the inductor resists the current change by changing its polarity and adds voltage to the source voltage. The increase in voltage forward-biases the diode and the energy is transferred to the output capacitor. The voltage level on the capacitor is now the source voltage plus the voltage over the inductor. See figure 2.12 for the switch open state.

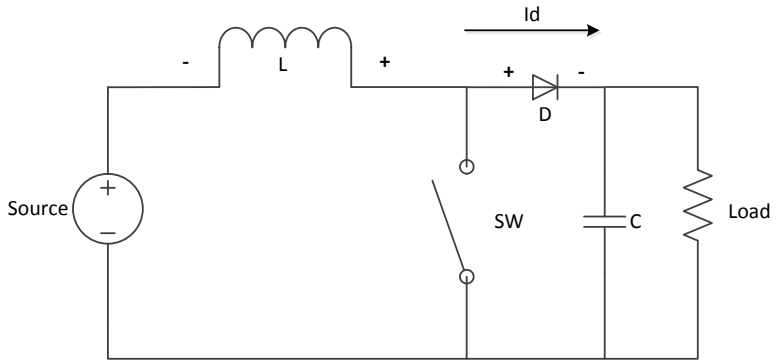


Figure 2.12: Boost regulator: Switch-open state, transfer energy to the capacitor (C).

2.5.2 Synchronous Rectifier

It is well known that switches are commonly implemented using MOSFETs. The switch used in the SPV1040 is based on a MOSTFET. The voltage at the MOSFET gate is used to turn the drain-source channel on and off. The use of MOSFET as diode is probably more unknown. This principle is called Synchronous Rectifier [8].

The synchronous rectifier act as a diode, the voltage at the anode needs to be higher than the cathode to conduct. A MOSFET is used to conduct the current. A controller determines whether the MOSFET should be turned on or off. There are two different ways of implementing the controller, active and passive controller. The active controller is typically using the same signal to control both the "switch" and the "diode" in the SMPS. The passive controller is sensing the voltage at the drain and source. The passive controller uses these voltage levels to determine whether the "diode" should conduct or not.

The benefit of using the synchronous rectifier instead of a Schottky diode is a lower forward voltage drop. A lower voltage drop will give higher efficiency and lower heat generation. The forward voltage drop of a Schottky diode is typically 0.3 V. The channel on-resistance of a MOSFET can be as low as few tens of $m\Omega$. As long as the current is not too high, the power wasted in a synchronous rectifier is significantly lower than a conventional diode.

2.5.3 SPV1040 charger

The SPV1040 is a quite new product and we have not been able to find a flight history of this chip. We know that other satellites plan to use the same chip, like NUTS from the Norwegian University of Science and Technology and the AMSAT Fox-1. The charging system is a critical system which is essential to mission success. The technology principle using switched-mode charging and MPPT algorithm has been used in satellites for many years, but SPV1040 has not been tested in space as far as we know. SPV1040 is using the CC-CV charging technique recommended by A123. The main benefits of using the SPV1040 are:

- High efficiency: We expect an efficiency of 85-90%.
- Low complexity: Both the charging system and MPPT is confined into a single chip which operates on its own.
- Small size: The SPV1040 is a small chip with a limited number of passive components needed.
- Redundancy: The SPV1040 chargers can operate independently of the rest of the power system.

2.6 Battery State of Charge Monitoring

It is important to keep track of the battery State of Charge (SoC) in a satellite power system. If the SoC is becoming lower than a predefined level, the satellite should be put in a power-save mode to avoid over-discharging and possibly damaging the batteries. The battery should be allowed to charge up to a predefined level and the satellite can resume normal operation again. There are several methods available on how to monitor the battery SoC.

2.6.1 Voltage Level

A capacitor has a linear relationship between voltage and SoC. A perfect battery would have a constant voltage regardless of the SoC. In real life the battery voltage will vary like something in between the capacitor and an ideal DC-source. This means that the battery voltage will give some information about the SoC. The accuracy is somewhat limited due to the non-linearity, in addition the battery voltage dependant on the temperature. Voltage measurement is a fast and simple way of getting a rough estimation of the SoC.

2.6.2 Coulomb Counting

Coulomb is the SI unit of one continuous ampere for one second ($C = As$). The battery charge capacity is often given in ampere hour (Ah), continuous current. A coulomb counter is measuring the current in and out of the battery through a shunt resistor. The shunt current is converted to charge during a fixed time interval and the resulting charge is added to a charge accumulator. The battery SoC is given as the SoC when the measurement started plus the result of the charge accumulator. This means we need to know or compute the SoC when the measurement starts.

Coulomb counting has several error sources. The coulomb counter does not take internal battery losses into account. Some of the energy drawn or put into the battery will be converted to heat and the coulomb counter measure external currents only. Batteries suffer from a certain amount of self-discharge. This is only a problem if there is a long time, like weeks or months, between the measurements. The coulomb counter time interval is important because you need a short time interval to capture short current transients. The maximum capacity of a battery tends to decrease as a function of charge cycles. One charge cycle is defined as a complete discharge from 100% and charge back to 100%. If the coulomb counter does not take this into account, the precision will be gradually lower with each charge cycle. There are additional error sources like offset errors, noise etc. See figure 2.13 for a graphical overview of coulomb counting.

2.6.3 Combined Measurement

A combined strategy is often applied to the SoC determination. Both voltage, temperature, coulomb counting and maximum charge is measured to compute a precise SoC. Different algorithms are used to take all these measurements into account. Dedicated ICs available do some or all of these measurements.

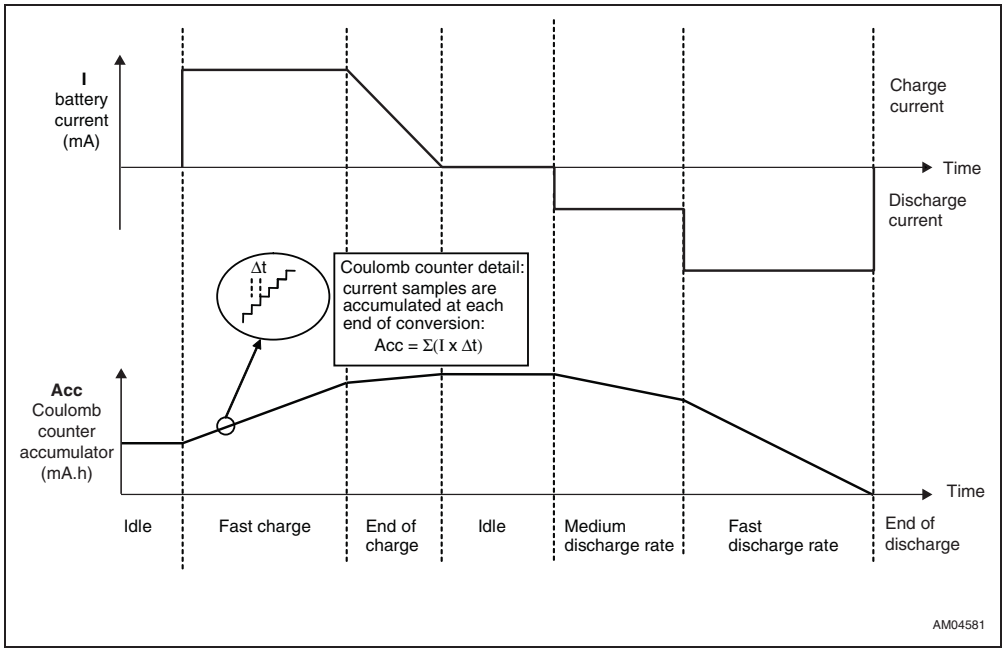


Figure 2.13: Coulomb counting principle. The current is measured at fixed intervals and converted to charge (mAh) [22].

Battery calibration is recommended by many manufacturers of battery powered devices like laptops and mobile phones. Battery calibration means a full discharge and charge cycle of the battery. The purpose of this is to re-establish the 100% level of the SoC. This can be a problem in space, because the battery will probably never be fully charged or discharged.

2.6.4 Current Sensing

Current sensing is important in power system monitoring. Standard ammeters are connected in series between the source and load. These ammeters will cause a power loss due to the voltage drop over the sensing resistor. Hall sensors are another possibility which measures the current indirectly. We know from physics that a current through a wire is setting up a magnetic field around the wire. The hall sensor is sensing this magnetic field and determines the current from the strength of the magnetic field.

The most common way of sensing current is using a shunt resistor in series between the source and load. The shunt resistor is a small resistor, in the mΩ range. The voltage drop over the shunt resistor is measured and the current through the shunt can be computed from Ohms law.

There are two methods of connecting the shunt resistor: High-side and Low-side, see figure 2.14. Both methods have advantages and disadvantages. The Low-side shunt should be avoided if you cannot tolerate a slight shift in ground potential. The High-side shunt will not disturb the ground potential but there can be a problem with large

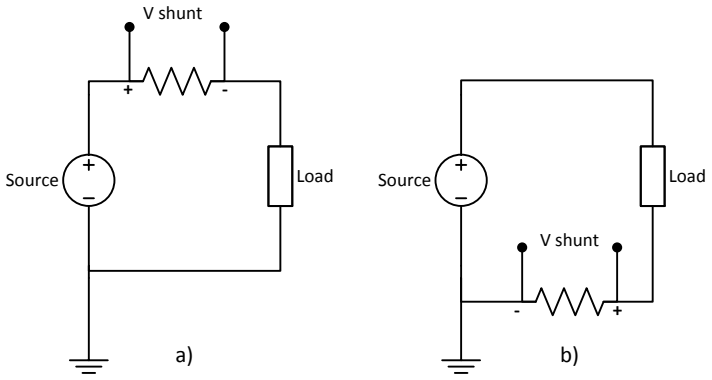


Figure 2.14: a) High-side current sensing. b) Low-side current sensing. The current is found from V_{shunt}/R_{shunt}.

common-mode voltage. The maximum common-mode voltage is determined by the maximum rating of the op-amp measuring the shunt voltage.

2.7 Space Related Challenges

When the satellite is deployed in space there are limited options of correcting errors. In space, the environmental conditions are quite harsh compared to the Earth's surface. We have to consider several environmental issues when designing the satellite.

2.7.1 Radiation

Outer space is filled with background radiation and radiation from stars. The Earth are protected from this radiation by the Earth's atmosphere and magnetic field. The CubeSTAR will be deployed in a Low-Earth Orbit (LEO), thus CubeSTAR will be less exposed to heavy ion radiation compared to deep-space missions and high earth orbits. Rad-hard components are commonly used in critical parts of space vehicles. The CubeSTAR will be designed using COTS components. This is mainly due to cost, because rad-hard components are very expensive.

Joakim Myrland is responsible for the main processor system, the OBC [14]. Myrland did a lot of research on these radiation challenges. We are mainly concerned about two different single event effects which can happen to a semiconductor in LEO.

The first is called Single Event Upset (SEU). SEU is a soft error, i.e. a bit-flip in a register or memory cell. If the SEU occur in a critical part, like processor data memory, the system might change behaviour or get stuck in a forbidden state. These errors can be solved be resetting the system, thus re-initializing the processor and rewriting flipped bits. It is more critical if the program code itself is corrupted. SRAM is often used as data memory, while the program code often is stored in flash memory. Myrland states

that the flash memory on the microcontrollers in use on CubeSTAR is more radiation tolerant compared to the SRAM.

The other event is called Single Event Latchup (SEL). SEL is a hardware error, e.g. a CMOS transistor can latchup if hit by radiation. A SEL cause a transistor to continuously conduct current, like a short circuit. This is a more severe error because the transistor can be permanently damaged. A power-toggle is needed to correct a SEL.

2.7.2 Triple Modular Redundancy

We have discussed that SEUs can change bits, e.g. a sensor reading can be corrupted. A common way of dealing with this problem in space applications is to use a Triple Modular Redundancy (TMR) scheme. TMR is implemented using three separate and redundant systems, e.g. three sensors instead of one. If one of the sensor readings becomes corrupted, the majority of the sensors will give the correct result. A voting system, often implemented on a processor, decides which result is valid. This method cannot detect several malfunctioning devices at once. This method increases system complexity so it should only be consider used in critical parts of the system.

2.7.3 Thermal Control

Heat can be transferred from one body to another using three different mechanisms: Thermal conduction, thermal convection and thermal radiation. Thermal conduction needs physical contact to transfer heat. Thermal radiation is transfer of heat through electromagnetic radiation, like the Sun transfers heat to the Earth. Thermal convection is transfer of heat from one place to another using gas and liquids, e.g. hot air rising. As a consequence of this, there is no thermal convection in space because of the vacuum.

This has to be considered when designing the satellite. Components that generate a lot of heat must have a way of getting rid of the heat. The best way of doing this is by thermal conduction, i.e. transferring the heat from the component to other parts of the satellite by physical contact. Components that cannot be too cold should be heated using an active heater or insulated from outer space, i.e. reducing the heat transfer to outer space using thermal radiation. One layer of insulation will reflect some of the radiated heat back to the source reducing the total heat radiated into outer space. Several layers will reduce the transferred heat even more.

Chapter 3

System Design

In this chapter the power budget of CubeSTAR will be estimated. All the hardware components chosen and the circuit schematic design of the EPS will be described. The requirements regarding the system design was discussed in chapter 1. In addition the firmware implemented on the MCU will be described. The block diagram in figure 3.1 describes the top level structure of the EPS design.

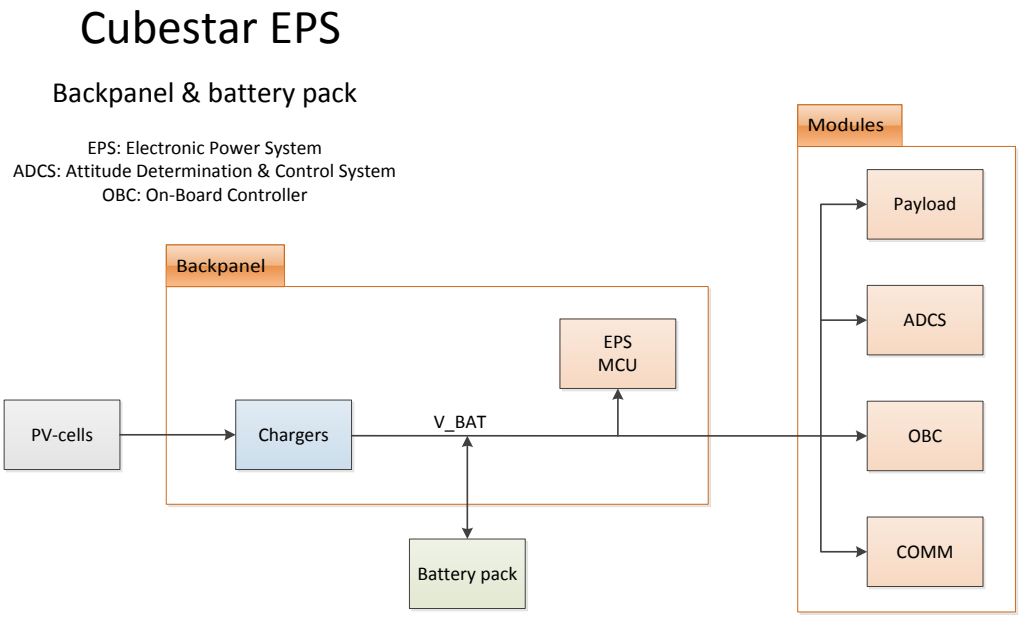


Figure 3.1: Top level EPS block diagram.

3.1 Power Budget

The power budget tells us whether the power system is sufficient for operation or not. The equation is simple: The power harvested during one orbit must be larger than the power used during one orbit. If not, the battery will be gradually drained until it is empty. The Orbit Average Power (OAP) depends on several parameters:

Orbit
The orbit determines the ratio between the eclipse and the sunlit portion of the orbit. In extreme cases you can end up with sunlight or eclipse during the complete orbit. We do not know the actual orbit parameters yet, so for our calculation we assume that 2/3 of the orbit is to be sunlit and the eclipse is 1/3 and we assume an orbit time of 90 minutes.

Solar Irradiation
The Air Mass zero (AM0) solar constant is the solar irradiation above the Earth's atmosphere. The manufacturer of the PV-cell use a solar constant of 1353 W/m^2. The CubeSTAR will be launched into a LEO which is not completely outside the Earth's atmosphere. Reflections in the atmosphere will give a slightly lower irradiation. We need to know the exact height of the orbit to compute the correct irradiation.

Albedo and Infrared shine
The Earth's surface will reflect some of the sunlight [28]. This reflection is called albedo and the amount of reflection depends on the colour of the surface. We know that dark surfaces have a low reflection coefficient and light surfaces have a high reflection coefficient. This means that ice and snow reflect much sunlight but dark surfaces like the ocean will not reflect much sunlight. The average albedo is said to be around 30% of the Solar irradiance. The Earth must maintain thermal equilibrium. Energy that is not reflected in the atmosphere or at the surface is absorbed. This absorbed energy is re-emitted as infrared light. The albedo gives an irradiation of approx. 410 W/m^2 and the infrared shine is approx. 240 W/m^2 [13].

Satellite Tumbling
When the satellite is released from the launch vehicle, the satellite will be free tumbling. Every time the satellite passes one of the magnetic poles, the magnetic field suddenly change giving the satellite a push. The ADCS will stabilize the satellite in a fixed orientation. We do not know yet how well the ADCS will perform. To calculate a correct power budget we need to know the satellite orientation.

Estimation
Because we do not know all the parameters needed we must estimate the power budget. A rule of thumb from Clyde Space tell us how to estimate the OAP for a satellite with four PV-cell panels in LEO: 60% of the total power from one side panel [7]. In our case

this means $0.6 \cdot 4 \cdot V_{MP} I_{MP} = 0.6 \cdot 4 \cdot 2.35\,\mathrm{V} \cdot 0.434\,\mathrm{A} = 2.45\,\mathrm{W}$. If we assume a power system efficiency between 80 - 90%, the OAP is approx. 2 W. This estimate is concurring with the calculations done by Oredsson [15]. Albedo and infrared shine is not included in this calculation. Clyde Space states that albedo is a bonus not a certainty and the germanium junction of the PV-cell is effective up to around 1800 nm, so only a small band of the infrared light is usable [7]. As long as we do not use more than 2 W average power during orbit, we will have a positive power budget.

3.2 Solar Power Charging System

There are several options on how to configure the charging system. The PV-cells can be connected in series, parallel or a combination. The V_{MP} of the PV-cell is 2.35 V and the output of the chargers is fixed to 3.6 V. This means that the PV-cells should be connected in parallel because the charger is a boost regulator and the input voltage must be lower than the output voltage. To get the 3.6 V charger output, the chargers must be connected in parallel. The charger and PV-cell configuration is shown in figure 3.2[1].

3.2.1 Charger and PV-cell Configuration

In our opinion there are two usable charger configurations: Two chargers in parallel or four chargers in parallel. Eight chargers in parallel will occupy too much PCB real estate. From the charger datasheet we can see that the efficiency depends on the input current. The maximum current allowed into a charger is 1.65 A. The maximum current from the PV-cells is when the radiation angle of incidence is zero. When the angle of incidence is zero, only one of the side panels are irradiated. From data in the PV-cell datasheet, we can compute the maximum current into the charger. The first option is four cells in parallel:

$$\begin{aligned} I_{MP} &= 4 \cdot 16.3\,\mathrm{mA/cm^2} \cdot 26.62\,\mathrm{cm^2} \\ &= 1736\,\mathrm{mA}. \end{aligned}$$

This is above the maximum rating of the charger and the charger efficiency will be lower if the input current is too high. This eliminates the option of two chargers in parallel. The next option is two PV-cells in parallel:

$$\begin{aligned} I_{MP} &= 2 \cdot 16.3\,\mathrm{mA/cm^2} \cdot 26.62\,\mathrm{cm^2} \\ &= 868\,\mathrm{mA}. \end{aligned}$$

This is preferred due to the increased efficiency and maximum ratings. In order to ensure that only two PV-cells in parallel are maximally irradiated, two PV-cells from one side panel and two PV-cells from the opposite side panel has to be connected in parallel.

[1] PV-cell X+D is removed to make room for Remove Before Flight (RBF) pin and integration connector.

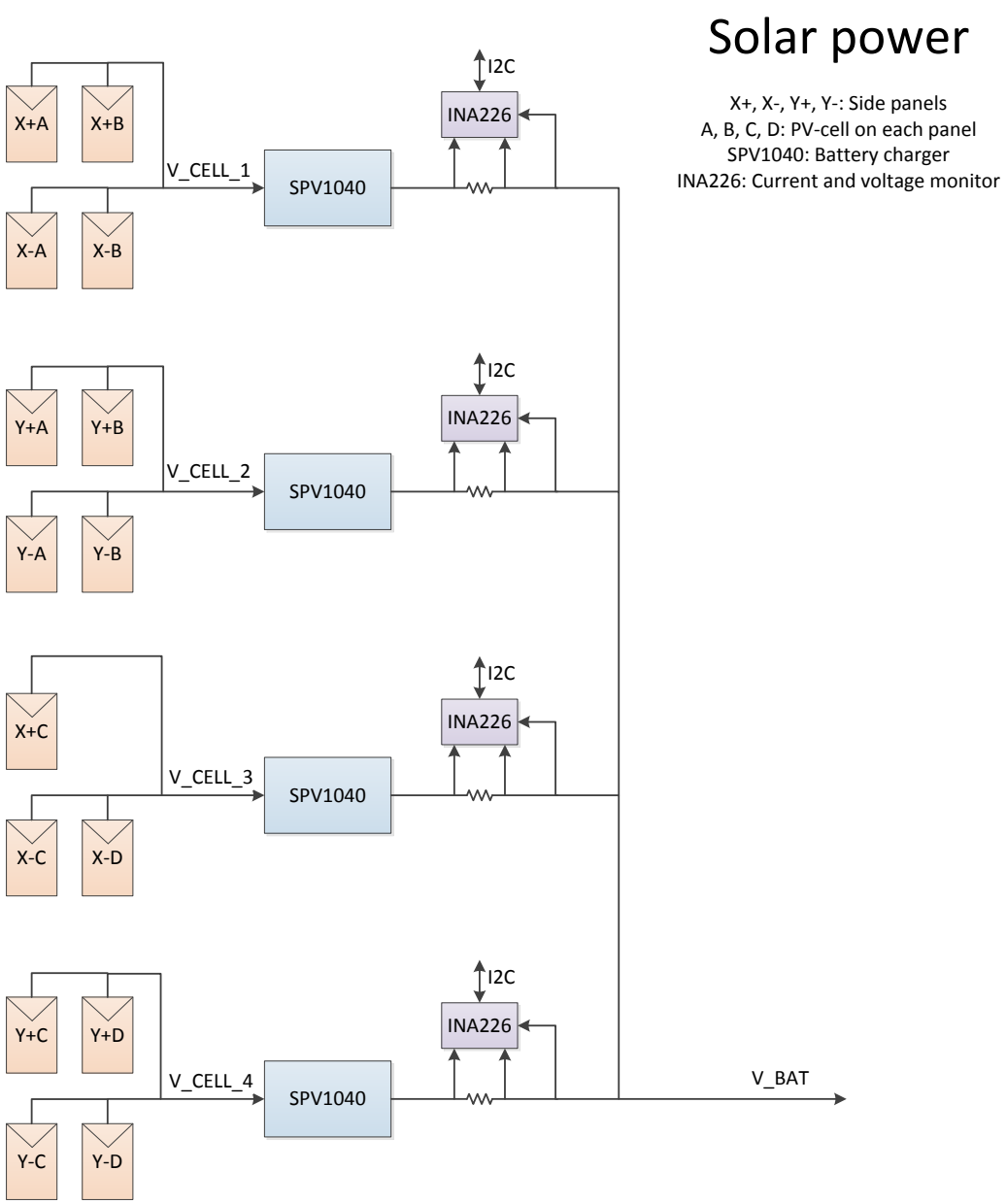


Figure 3.2: Solar power block diagram.

This ensures that we will get power regardless of the satellite orientation with respect to the Sun. Another benefit with four chargers is that if we lose one charger, we only lose a quarter instead of half the input power.

Four INA226s is implemented to measure the current from each charger. Each charger should receive some power during orbit. The charger current will give us information if a charger, or PV-cells connected to a charger, is malfunctioning. Another important task of these INA226s is battery voltage monitoring. More details about INA226 will be given later.

3.2.2 SPV1040 Configuration

The SPV1040 charger is configured using external passive components (figure 3.3). The

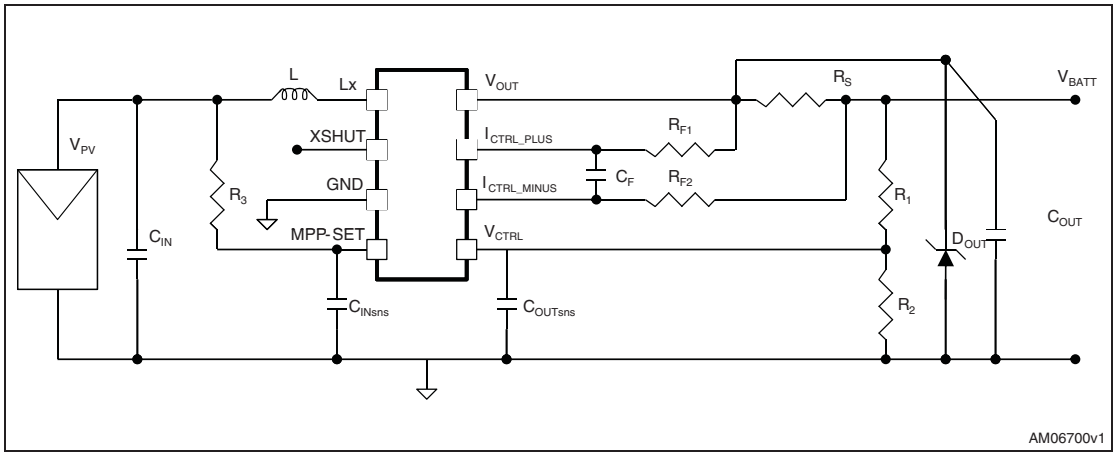


Figure 3.3: Passive external components of SPV1040 [23].

input and output capacitor (C_{IN}, C_{OUT}) must be large to reduce the voltage ripple on the input and output. The manufacturer recommends ceramic capacitors with low resistance for highest efficiency. 100 μF ceramic capacitors is chosen for both the input and output. The inductor size is dependent on the maximum voltage and current from the PV-cell connected to the input. With the PV-cell chosen for our mission, an inductor of 10 nH is found to be suitable. Low internal resistance of the inductor is important to achieve high efficiency.

To make the charging system independent of the rest of the system, the XSHUT pin is connected to the MPP-set pin. The chargers are enabled when the connected PV-cells are irradiated. There is a hysteresis built in on the XSHUT and MPP-set pin to prevent rapid on-off toggling.

The output voltage is monitored by the V_{CTRL} pin. The charge cut-off voltage is set with resistor R_1 and R_2. The maximum voltage of the CubeSTAR is 3.6 V. If we assume a worst-case scenario, R_1 must be set to 680 kΩ and R_2 to 392 kΩ to never exceed 3.6 V. All passive components are chosen according to the manufacturer recommendations [23].

3.2.3 Blocking Diode

In chapter 2 we saw that the PV-cells have a bypass diode installed. This diode is usable when we connect several PV-cells in series but in this design the PV-cells are connected in parallel. The PV-cells on opposite side panels are connected in parallel. This means that one PV-cell is irradiated while the other is in the dark. We model a PV-cell as a current source and a diode. If the PV-cell is shaded, the current is zero and the diode is forward biased. The shaded PV-cell will draw current from an irradiated PV-cell and from the battery. This means that we have to block current from going back into the shaded PV-cell. This is done by implementing a blocking diode. The blocking diode will be reverse biased if the corresponding PV-cell is shaded, see figure 3.4.

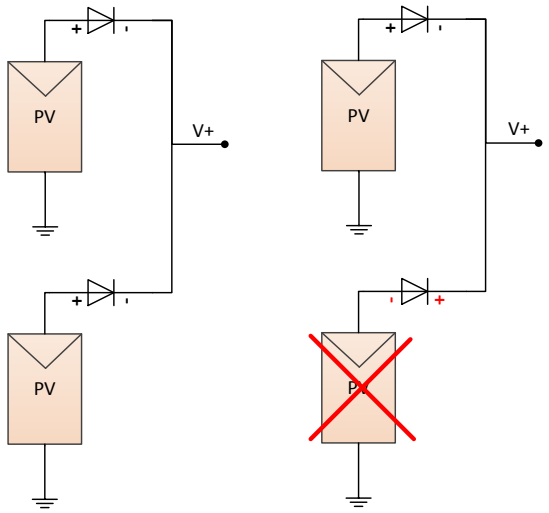


Figure 3.4: Blocking diode operation. The diode is blocking current into the shaded PV-cell.

A Schottky diode has a forward voltage drop of 0.3 V typically. This is a power loss we want to avoid. In chapter 2 we discussed the synchronous rectifier. It turns out that ST microelectronics has made a diode (SPV1001N30) based on a synchronous rectifier intended to be used on PV-cells. The manufacturer promise an average forward voltage drop of 70 mV at 5 A. Tests performed show a forward voltage drop of 53 mV at 1 A. This is very low compared to Schottky diodes, so these diodes are implemented in the EPS design.

3.3 Power Bus

The main task of the EPS is to provide power the satellite subsystems. Due to the variety of subsystem power requirements, it has been decided to distribute unregulated power. The EPS must be able to turn power on and off to each module. The OBC decides which module should receive power. The power bus must be current-limited in case of a short-circuit to prevent battery discharge and damage to components. An

automatic power cycling feature is implemented for fast removal of SELs. See figure 3.5 for the power distribution block diagram.

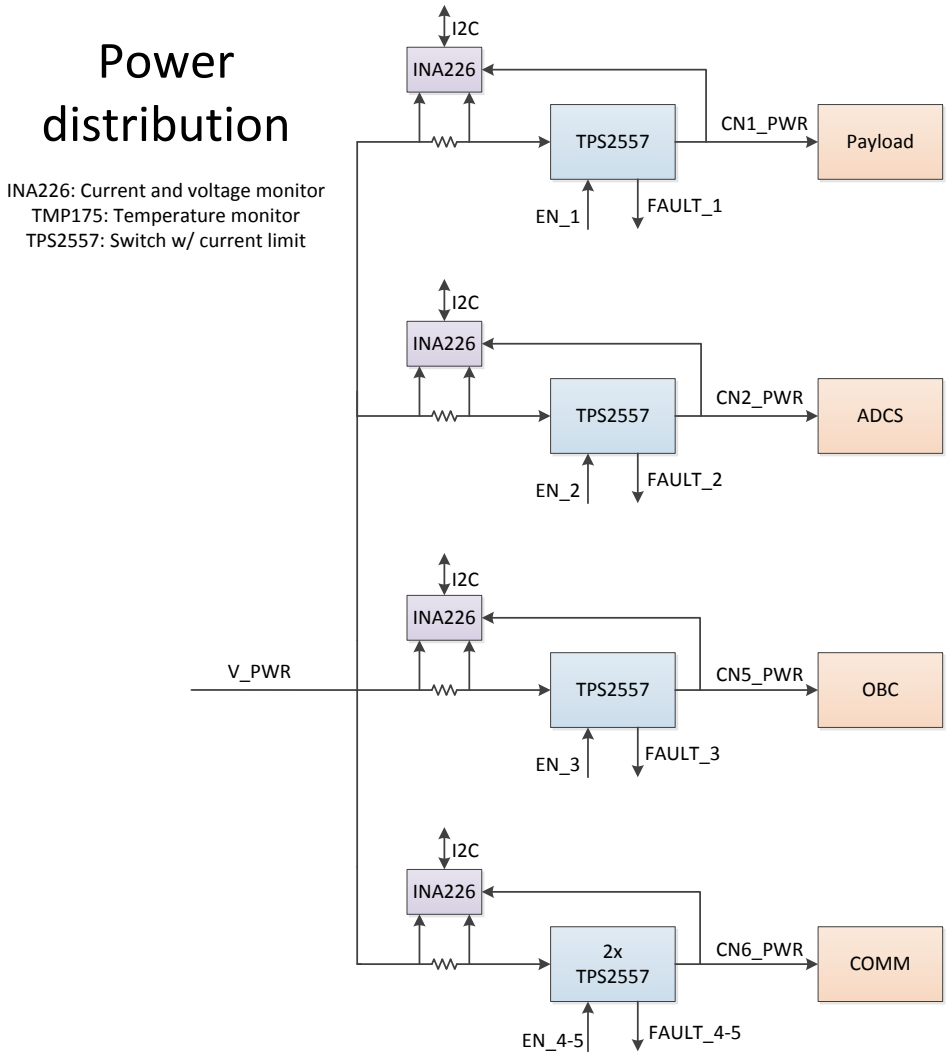


Figure 3.5: Power distribution block diagram.

3.3.1 Power Distribution Switch

The component chosen for the power distribution is the TPS2557 from Texas Instruments. TPS2557 is a current-limited switch with enable and fault signal. The current limit can be set with an external resistor in the range from 500-5000 mA. The on-

resistance of the switch is low, typically 22 mΩ. Each module on the satellite has different power requirements, so the current-limit should be tailored to suit the specific module. The enable signal is controlled by the EPS MCU. The fault signal will give the MCU information of an over-current or over-temperature event. To be able to quickly remove a SEL, the TPS2557 is configured in auto-retry mode. When the fault signal is asserted the switch will be disabled. The RC time constant, set by C_{retry} and R_{retry}, will determine the delay before re-enabling the switch (figure 3.6).

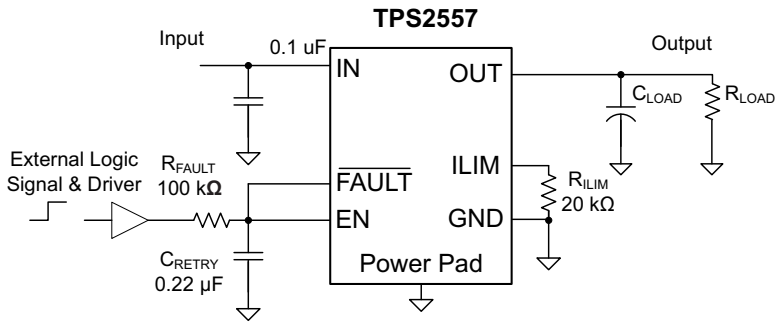


Figure 3.6: Power distribution switch with auto-retry [27].

The short-circuit current caused by a SEL will vary in size. If the current is not high enough to trigger the over-current protection of the TPS2557, the SEL can be detected and removed by a power-toggle command from the OBC or the ground station.

To ensure the modules are truly disabled during a power-toggle, we chose to implement a pull-down resistor on the output of the TPS2557. If not, we can risk that the switch is toggled but the voltage level during the toggling is high enough to keep the module alive. The value of the pull-down resistor has to be high to avoid a large leakage current thus wasting precious power. A resistor value of 100 kΩ has been chosen. The worst-case toggle time of the switch can be computed when knowing the pull-down resistor value and the total capacitance on the output of the switch.

If the EPS MCU fails, the enable signals to the power distribution switches should be tied to a default state using pull-up and pull-down resistors. It has been decided that the OBC and the COMM is default enabled, while the ADCS and payload is default disabled.

3.3.2 Sensor Power

Brownout is the voltage level where a system is shut off. The satellite has a brownout level of 2.35 V because of the under-voltage lockout in the TPS2557. All sensors used in the EPS is guaranteed to operate down to 2.7 V. To be able to rely on the sensors all the way down to the brownout voltage, it has been decided to implement 3.3 V regulation for the EPS system. The TPS63001 regulator is chosen for this purpose. The reason for using this regulator is the high efficiency, up to 96%, and it has a power save mode

for high efficiency during low current consumption. In addition, it has a small count of passive components needed. The power to the sensors is enabled using a TPS2557 like the rest of the modules. If one of the sensors misbehaves, we have the opportunity to power-toggle the sensors.

3.3.3 Current and Voltage Monitoring

Current monitoring is important during the integration phase. It will be used to verify the power budget calculations. The current consumption on each module and the total current consumption of the satellite will be monitored. The EPS will monitor the different voltages on the power bus. In space it will give us information about unwanted behaviour and complement the information from the power distribution switches. The component chosen for current and voltage monitoring is the INA226 from Texas Instruments.

INA226 can monitor both voltage and current. INA226 features an on-board Analog-to-Digital Converter (ADC) plus an automatic conversion from shunt voltage to current feature. The input range of INA226 is fixed ($\pm 82\,\mathrm{mV}$), so the size of the shunt resistor has to be carefully selected. To minimize the power loss the shunt resistor should be low. To utilize most of the ADC range and minimize noise the shunt resistor should be high. The maximum expected current in the system is 1.5 A, which give a suitable shunt resistor value of $\frac{82\,\mathrm{mV}}{1.5\,\mathrm{A}} \approx 50\,\mathrm{m\Omega}$. To reduce the power loss a shunt resistor of 25 mΩ has been chosen.

The configuration and conversion result registers are accessed through the Inter-Integrated Circuit ($\mathrm{I^2C}$) interface. INA226 can be configured in many ways. Low current consumption and low noise is wanted. Some form of averaging is needed because we cannot transmit sensor data with high sampling frequency over the data link and the computational power of the MCU is limited. The manufacturer states that a combination of the longest conversion time and highest number of averages will give the highest measurement accuracy, depending on your systems timing requirements. We will gather sensor data every 10 seconds and the INA226 has been configured to average data over 1024 points. The conversion time of the voltage and current measurement can be set individually. The voltage is more stable compared to the current so a short voltage conversion time and a longer current conversion time is chosen. A voltage conversion time of 1.1 ms and a current conversion time of 8.244 ms is chosen, which give a total conversion period of $(8.244\,\mathrm{ms} + 1.1\,\mathrm{ms}) \cdot 1024 = 9568\,\mathrm{ms}$. This high conversion period will help with filtering measurement noise and find the average current from the chargers or to the modules. The voltage measurement is used to determine if the batteries are close to completely discharged. The averaging of the voltage is important to filter out temporary voltage drops due to large transient currents.

3.3.4 Measurement Accuracy

It is important to calculate the accuracy of the sensors to be able to find the uncertainty of the sensor data. When using op-amp based sensors, the input offset voltage (V_{os}) is

one of the largest factors affecting the accuracy. The input offset voltage is defined as the voltage you need to apply to the op-amp input to get zero output voltage. This offset will give a gradually higher error when the voltage or current we want to measure is lowered. The worst-case offset $V_{os(max)}$ is 10 μV for the current measurement and 7.5 mV for the voltage measurement on INA226. We can now calculate the worst-case input offset error

$$e_{V_{os}} = \frac{V_{os(max)}}{V},$$

where V is the voltage we want to measure. The Common Mode Rejection Ratio (CMRR) and Power Supply Rejection Ration (PSRR) specification of INA226 is quite good and is not thought to affect the measurement accuracy notably. The input offset voltage drift ($V_{os-drift}$) is depending on the temperature deviation (ΔT) from 25°C. We expect internal temperatures from 0° to 25°C. The worst-case $V_{os-drift}$ is 0.1 μV/°C for the current measurement and 40 μV/°C. We can compute the worst-case offset drift error from

$$e_{V_{os-drift}} = \Delta T \frac{V_{os-drift(max)}}{V}.$$

The gain error of INA226 is given directly in the datasheet. For the current measurement, the shunt resistor tolerance is a big contributor to the total error. A resistor with 1.0% tolerance has been chosen which adds directly to the measurement error. The Temperature Coefficient of Resistance (TCR) of the resistor will contribute to the measurement error. The chosen resistor has a TCR of ±70 ppm/°C and the error due to the TCR can be computed from

$$e_{TCR} = TCR \cdot \Delta T.$$

The total error is found using Root-Square Sum ($E_{tot} = \sqrt{E_1^2 + E_2^2 + ... + E_n^2}$) [18].

An Excel spreadsheet has been made where we can modify the different parameters and see how the measurement error changes. The total worst-case error of the voltage measurement is 0.25% and the typical error is 0.044% when measuring 3.3 V at 0°C, see table 3.1. We see that the input offset error is dominating and if we want to measure 2.5 V, the typical error increases a little bit to 0.055%.

Table 3.1: INA226 voltage measurement error, measuring 3.3 V at 0° C

Source	Worst-case error [%]	Typical error [%]
$e_{V_{os}}$	0.23	0.038
$e_{V_{os-drift}}$	0.030	0.0076
Gain error	0.10	0.020
Total	0.25	0.044

The total error of the current measurement is found in table 3.2 when measuring 100 mA at 0°C. We see that the resistor tolerance error is dominating. When measuring below 10 mA the input offset voltage will begin to dominate. If we want to measure 1 mA, the typical error will be as high as 10%. This is not critical because the processor in use is based on 8-bit architecture. Floating point variables are obviously not an option, so we discard the decimal part of all our measurements. The typical error of the current measurement will be 1.0% or ±1 mA, whichever is greater.

Table 3.2: INA226 current measurement error, measuring 100 mA with R_{shunt}=25 mΩ at 0° C

Source	Worst-case error [%]	Typical Error [%]
$e_{V_{os}}$	0.40	0.10
$e_{V_{os-drift}}$	0.10	0.020
Gain error	0.10	0.020
R_{shunt} tolerance	1.0	1.0
R_{shunt} temp. coefficient	0.18	0.18
Total	1.10	1.02

3.3.5 Current Sensor Filtering

When it comes to noise, the current measurement is more sensitive to noise than the voltage measurement. The low shunt resistor value results in a very low voltage drop which is easily contaminated by noise. The averaging of 1024 measurements provide some noise filtering, but the manufacturer of the current sensor is recommending a passive low pass filter on the op-amp input. When we add additional resistance at the input, we alter the bias currents of the sensor. This will increase the measurement error. Therefore the manufacturer of the sensor recommends a resistor value below 10 Ω. The capacitor value is recommended to be in the range 0.1 - 1 μF. A resistor value of 6.8 Ω and a capacitor value of 1 μF was chosen.

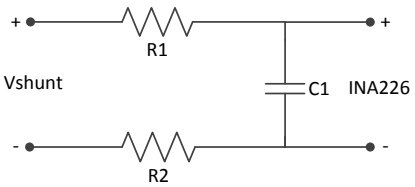


Figure 3.7: Passive LP-filter on current sensor input.

From figure 3.7 we see that the total impedance (Z_{tot}) of the filter is

$$\begin{aligned} Z_{tot} &= R + jX \\ &= R_1 + R_2 - j\frac{1}{\omega C_1}. \end{aligned}$$

The cut off frequency (f_c) is defined to be where $R = X_C$. With our values we can find the cut off frequency:

$$\begin{aligned} R &= X_C \\ R_1 + R_2 &= \frac{1}{\omega C_1} \\ f_c &= \frac{1}{2\pi(R_1 + R_2)C_1} \\ &= 11.7kHz. \end{aligned}$$

3.3.6 Langmuir Probe Release

The EPS is controlling the m-NLP release circuits. On command from the OBC, the EPS will enable power to the release circuits. There are four separate release circuits, one for each probe. The probe is held in place by a thin thread and the thread is resting on two resistors. When the release circuit is enabled, we send 0.5 A through the resistors causing them to heat up. When the temperature is high enough, the thread melts and the probe is released. Because the resistors are drawing a lot of current we choose to release one probe at a time. When the probe is released, a switch that monitors the release status changes state. The EPS is monitoring the state of the switch and the OBC can obtain information about which probe is released and not.

3.4 Battery Pack

The battery pack is an important part of the EPS. Without the batteries the operational capability of the satellite is severely reduced. The batteries proposed for this mission is $LiFePO_4$ cells from the manufacturer A123. It has been chosen to implement six of the A123 batteries in parallel. In normal operation one or two batteries should be sufficient, but in case of a catastrophic failure of the charging system the redundant cells ensure extra mission time. If the charging system is overheated it will need time to cool down, hence extra battery time is needed. Another argument for oversizing the battery pack is that the satellite needs extra power during the start-up sequence for detumbling, antenna and m-NLP release.

It has been discussed to split the batteries into two packs of three cells, but to keep things simple we have decided to stay with one pack of six cells. As mentioned in chapter 2, the batteries can tolerate much abuse without malfunctioning. We believe that other errors than battery failure is more likely to cause mission failure. Two battery packs would demand more of the control system and we want to keep things as simple as possible. See figure 3.8 for a block diagram of the battery pack.

3.4.1 Temperature Sensor

TMP175 from Texas Instruments is chosen for monitoring temperature. TMP175 has a typical accuracy of $\pm 0.5°C$ and a worst-case accuracy of $\pm 1.5°C$. The TMP175 conver-

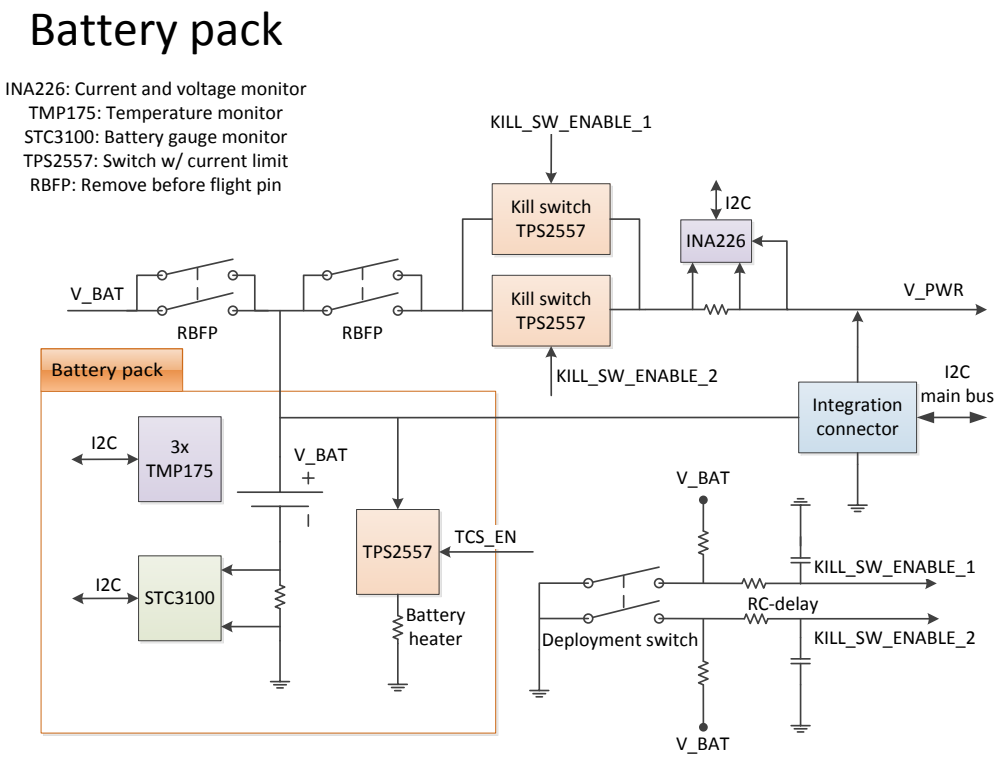


Figure 3.8: Battery pack block diagram.

sion registers are accessed using the I^2C interface. The temperature is changing slowly and in order to save power the TMP175 is configured in triggered mode. Each time the housekeeping task starts, the TMP175s are triggered to do one measurement and return to power down mode after conversion.

3.4.2 Battery Monitoring

If the EPS is designed properly and the satellite is functioning as expected, the satellite should operate without any need for interference. To be able to recover from an erroneous power condition it is important to implement battery monitoring. There should be both firmware and hardware routines ensuring battery monitoring in case of processor failure. Three different conditions regarding the battery state is defined: Low battery voltage, low battery temperature and satellite brownout.

Low Battery Voltage

If the battery voltage becomes lower 2.8 V and the temperature is normal, the battery is near completely discharged. Then the EPS will issue a battery warning and the OBC will

start the shut-down procedure of the payload, ADCS and COMM. The OBC and EPS power requirements are negligible compared to the other systems, so they will continue to operate. The EPS is monitoring the battery voltage and when the battery voltage reach 3.3 V, the battery warning is cleared. The INA226s used to measure charger current is used to monitor the battery voltage in addition.

Low Battery Temperature
The battery voltage and capacity is dependent on temperature. If the temperature is too low, the battery voltage will sink below the minimum voltage (2.8 V) for normal operation. Another concern is that the usable capacity of the battery is reduced if the temperature is too low. After testing we found that the thermal control system should be activated if the battery temperature becomes lower than -7°C. The thermal control system is deactivated when the battery temperature reach -5°C.

Satellite Brownout
If the processor running the firmware routines fails, a hardware routine should shut off the satellite in case of dangerously low battery voltage. The TPS2557 switches used as kill switches in the satellite have a under-voltage lockout threshold at 2.35 V. This means that all of the satellite will be shut off except the battery charging system which runs independently. This will allow the batteries to recharge until the kill switches are re-enabled. The battery voltage will still be too low for normal operation. The EPS will monitor the battery voltage and clear the battery warning when reaching 3.3 V. See table 3.3 for a summary of the different battery conditions and actions.

Table 3.3: Summary of battery conditions and corresponding actions

Condition	Action
Voltage below 2.8 V and normal temperature	Issue battery warning
Temperature below -7°C and no battery warning	Activate battery heater
Voltage below 2.35 V	Global satellite brownout

Coulomb Counter
A battery SoC measurement has been implemented just for informational purposes. The SoC measurement can give the ground station crew a heads up in case a battery warning is imminent. The SoC measurement will complement the other sensor data regarding the battery status. We discussed several methods of battery SoC measurement in chapter 2. If we used normal lithium-ion batteries, the voltage measurement should give good enough results for our application. The flat discharge curve of the $LiFePO_4$ battery cell makes it difficult to measure the SoC accurately using voltage measurement. For the state of charge monitoring, STC3100 from ST Microelectronics has been chosen. This IC will continuously measure the current in and out of the battery pack and compute the battery SoC. This is a much better option than doing the calculations using the

MCU. The MCU has limited processing power and the STC3100 is only using 300 μW during operation. The SoC is stored in an accumulator which is accessed using the I^2C interface. Because battery capacity most often is stated in mAh, not coulombs, the output of the STC3100 is in mAh.

The measurement accuracy is dependent on the quality of the clock signal used by STC3100. The worst-case error is 3.5% using the internal clock, the typical error is not stated in the datasheet. This error is quite high dominating over the other error sources. To increase the accuracy, an external clock signal can be applied decreasing the worst-case error to 1%. We do not need a very precise SoC, so to keep things simple the internal clock is used. The error of the shunt resistor is not included in the previous figures, they have to be added using the root-square summation discussed earlier. See table 3.4 for the total worst-case error of the coulomb counter.

Table 3.4: Total worst-case error of coulomb counter

Source	Error [%]
Coulomb counter (internal)	3.5
R_{shunt} tolerance	1.0
R_{shunt} temp. coefficient	0.175
Total	3.64

3.4.3 Depth of Discharge

The total capacity of the six cell battery pack is 6600 mAh. We assume that one orbit is 90 minutes long and the eclipse is 1/3 of the orbit. An OAP of 2 W has been estimated. If we use all available power, it equals an average current consumption of around 600 mA. 600 mA for 30 minutes equals 300 mAh. In other words, the battery pack will be discharged $\frac{300}{6600} \approx 5\%$ during each eclipse.

If the battery pack is to become completely discharged, the satellite will be put in battery warning mode with only the EPS and OBC enabled. This means almost all power from the PV-cells is used to charge the batteries. From the OAP we find that the batteries are charged with 600 mA average current or 600 mA $\cdot$ 1.5 h = 900 mAh each orbit. Thus, it will take $\frac{6600}{900} = 7.33$ orbits to fully charge the battery pack.

3.4.4 Thermal Control System

We distribute unregulated power in the satellite. If the battery voltage becomes lower than the brownout level of 2.35 V the satellite will shut down. The battery voltage and capacity is dependent on temperature which means that thermal control is important. A thorough thermal analysis of the CubeSTAR has not been performed, but we can check flight data from other similar cubesat missions to find out more about the expected temperature range.

The Swisscube implemented an active Thermal Controls System (TCS) of 0.5 W which is enabled at -3°C and disabled at +2°C. Flight data available[2] show that the battery temperature is varying from -3°C to +15°C. This tells us that TCS is working as expected, limiting the minimum temperature to -3°C. Flight data from the Cal Poly 3 cubesat show approximately the same battery temperature range, from -8° to +15°C [10]. We have not been able to find out if Cal Poly 3 implemented a TCS. Clyde Space, a commercial manufacturer of cubesat battery packs, implements an active TCS. They implement a 0.44 W active TCS, which is enabled between 0° to 5°C [20]. Other satellites like AAUSAT 1 and 2 did not experience sub-zero temperatures. We have talked with an associate professor at Aalborg University and they did not implement any battery heating system nor had any problems with low temperatures.

Either way these flight data show that battery temperatures below 0°C can be experienced, depending on the orbit parameters. In addition the battery cell used in our mission is expected to have a low degree of self-heating due to low internal resistance. Because we cannot be exactly sure how low the battery temperature will fall, an active TCS of 0.5 W has been chosen. The battery heating system will be default off and the ground station crew can enable the heating system and adjust the thresholds for enable/disable while deployed in orbit. When the heating system is on, the heating system will regulate itself according to the enable/disable thresholds.

The battery pack is made up by three batteries on each side of the PCB. It has been chosen to divide the heating power into four 82 Ω resistors in parallel, two on each side, to distribute the heat across the battery module. It is important that the temperature sensors are located in some distance away from the heating resistors to avoid measuring the best case temperature. The TCS is controlled by the EPS using a TPS2557 switch. To avoid unnecessary waste of power, the TCS is set to default off in case of a processor failure. The TCS will never be enabled during a low battery voltage warning. During a battery warning, only the EPS and OBC is enabled. They do local voltage regulation, so they can operate all the way down to the satellite brownout regardless of the battery temperature.

In vacuum there is no thermal convection. We have to ensure that the heat from the heating resistors is transferred to the batteries. The way to accomplish this is to transfer heat by conduction which means physical contact. We have discussed several approaches to this problem. First, thermally conductive potting was considered. It turned out to be difficult to find space qualified potting with high thermal conductance. It is necessary to use space qualified adhesives and potting to comply with outgassing requirements of the launch provider. Otherwise we can risk the satellite being rejected by the launch provider. The next alternative, which was chosen, is to transfer the heat to the power and ground layer of the PCB. The battery terminals are connected to these layers, thus transferring heat to the battery cell core. Vias are used to increase the thermal conductance between the heating resistors and the inner PCB layers.

Passive thermal control of the battery pack has been considered. Passive thermal control is most often implemented using some kind of insulation. For deep space missions,

[2]Swisscube flight housekeeping data: `http://ctsgepc7.epfl.ch`. Accessed: 28.09.2012

the space vehicle is often insulated using multi-layer insulation. CubeSTAR do not have room for multi-layer insulation because almost all of the outer area is covered with PV-cells. Inside, around the battery pack, there are difficulties with finding a way to insulate. First, we need to find a material that does not conduct current or is out-gassing in vacuum. Second, there are problems finding a way to mechanically implement the insulation. The battery pack is located inside at centre of the satellite. As we discussed in chapter 2, we know that PCBs with solid power and ground layers will insulate against thermal radiation. This knowledge, combined with the difficulties of implementing the insulation, has helped to decide to not implement passive thermal control.

3.4.5 Mechanical Fixing

The battery cells are quite heavy, 40 grams, compared to the other components on the satellite. Due to the high acceleration during launch it is important that the batteries are securely fastened to the PCB. All adhesives and coatings must be space qualified due to outgassing requirements. The NASA space qualification requirement[3] is: Total Mass Loss (TML) less than 1% and Collected Volatile Condensible Materials (CVCM) less than 0.1%. We have tried to find out what other satellites has done to fasten the batteries. Aalborg University in Denmark has used 3M Scotch-Weld EC-2216 gray with success. This is a space qualified 2-part epoxy adhesive with a TML of 0.77% and a CVCM of 0.04%. This adhesive is recommended to securely fasten the batteries to the PCB.

It is important to isolate the battery terminals and exposed part of connectors electrically. We cannot risk to short-circuit the batteries accidentally because each battery cell can supply 30 A of continuous current. The same adhesive, EC-2216, can be used to isolate the battery terminals. Another alternative is Dow Corning 93-500, which is a space grade encapsulant with a TML of 0.14% and a CVCM of 0.01%. DC 93-500 has lower viscosity, which means that we probably need a mould. The advantage of using lower viscosity is that we can achieve a thin and uniform coating layer.

3.4.6 Kill Switch and Remove Before Flight Pin

From the cubesat standard we find some requirements regarding power during launch [5]. The battery pack should either be completely discharged before launch or removed from the circuit using a kill switch. The reason is to avoid accidental release of antennas and probes before the satellite is released from the Poly Picosatellite Orbital Deployer (P-POD) and prevent electrical and RF disturbance towards the launch vehicle. It will take quite a long time, seven orbits, to fully charge the batteries from an empty state. If the charging system suffers from a catastrophic failure during launch the satellite will never be operational. Therefore, a kill switch is implemented to remove the fully charged batteries from the circuit during launch.

[3]Online database with outgassing data for spacecraft materials: `http://outgassing.nasa.gov` Accessed 06.02.2013.

The kill switch is implemented using two stages, called the deployment switch and the kill switch. The deployment switch is a mechanical switch which is operated by two spring loaded pins which are integrated into the satellite structure. When the satellite is inserted into the P-POD the pins will operate the deployment switch, disabling the kill switch. When the satellite is released from the P-POD the springs will release the pins and the deployment switch will enable the kill switch. The reason for using two stages is that we do not want the load current to run through a mechanical switch. A MOSFET is considered to be more reliable. The TPS2557 with auto-retry functionality is implemented as kill switch. To delay the start-up a few seconds after being released from the P-POD, a RC-circuit on the enable signal from the deployment switches is implemented. The delay is set to 5 seconds.

The RBF pin is operating another switch removing the battery from the circuit. The purpose of this switch is to shut off the satellite and avoid battery discharge during storage. After delivering the satellite to the launch provider, there might be a delay of several months before launch. The RBF pin must be operated after the satellite is placed into the P-POD and the kill switch is disabled. See figure 3.8 for the implementation of the switches.

3.4.7 Integration Connector

An integration connector is implemented to give access to the satellite after assembly. The integration connector allows you to charge the batteries, power the satellite from an external power supply and give you access to the main I^2C bus. During the start of the integration phase, it is important to be able to power the satellite using a lab supply with current limit. The batteries can give up to 30 A continuous current each, so hardware errors like short circuits can trigger catastrophic events. The integration connector will give you debugging access and the possibility of battery charging after the satellite is fully assembled.

3.5 Microcontroller Hardware

The EPS is controlled by a MCU. The MCU configures the sensors and gather all the sensor data using the I^2C interface. It will operate the power distribution system and respond to a set of commands issued by the OBC. The MCU chosen for the EPS is Atmel XMEGA128A1U. The reason for using this MCU is that XMEGA128A1U has all the hardware features we need and the OBC and COMM are using the same MCU as well. It has four I^2C peripherals and we need three, which is difficult to find in smaller MCUs. In addition there is a lot of I/O pins on the XMEGA128A1U which we need for all the control signals. First we will look at the hardware implementation of the MCU, then the firmware structure in the next section. See figure 3.9 for a block diagram of the MCU.

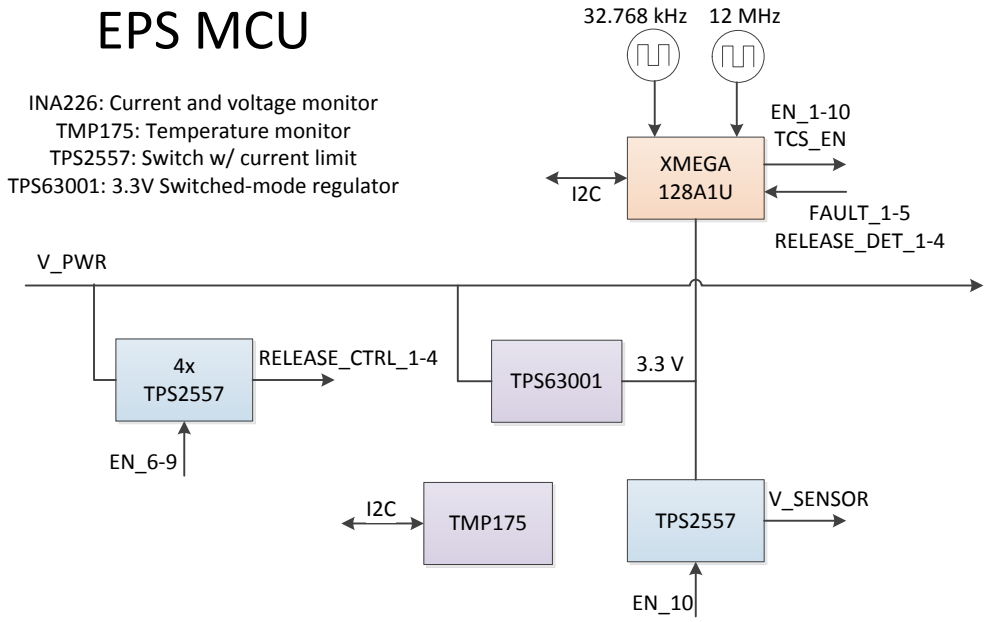


Figure 3.9: Microcontroller block diagram.

3.5.1 Oscillators

The MCU will need two oscillators, one running the system clock and one running the Real-Time Counter (RTC). The system clock is used by the MCU for general operation and the speed chosen for this clock is important. The manufacturer states that the maximum clock rate is determined by the supply voltage. A 3.3 V regulated power supply is implemented for the EPS, so the clock rate that yields best results can be chosen. After testing 12 MHz was found to be a good choice for the system clock.

There are several options when it comes to selecting source for the system clock. To be able to freely choose between clock rates, the 2 MHz internal oscillator with the Phase-Locked Loop (PLL) was chosen. The PLL can increase the clock rate with factors from 1x to 31x. The satellite will experience a wide range of temperatures. To provide temperature drift compensation and improved accuracy on the 2 MHz internal oscillator, a digital frequency locked loop (DFLL) is used.

In order to run the RTC and provide a source for the DFLL, we need a 32768 Hz oscillator. The RTC need a high precision oscillator to minimize the drift. The internal 32768 Hz oscillator of the XMEGA128A1U has an accuracy of $\pm 0.5\%$ which is not good enough. An external crystal has been implemented with an accuracy of ± 20 parts per million, which equals $\pm$ 0.002%. This is good enough for our application. To improve

the temperature stability on the crystal, we use class 1 (C0G) capacitors which has a very low tolerance and temperature drift. The RTC will be used to wake the MCU from sleep to start the housekeeping task every 10 second. Another task of the RTC is to increment an uptime counter. After the satellite is released from the P-POD, we need to wait 30 minutes before releasing the antenna and the m-NLP and start transmitting beacon. The OBC will gather the uptime value from the EPS and compare it with his own uptime counter.

3.5.2 Power Reduction

In addition to keep the system clock rate low there are several other things we can do to minimize the power usage. All peripherals not in use should be disabled. This means disabling peripherals like ADCs, digital to analog converters, analog comparators and serial peripheral interface. All pins not in use should be tied to a high or low state using pull-ups or pull-downs because CMOS is drawing current in the switching moment only. A floating pin could rapidly toggle between high and low state, thus drawing a lot of current.

There are several sleep modes available. The EPS must be able to wake up from RTC interrupt, I^2C command received from OBC and interrupt from the fault signals of the TPS2557. This limits the sleep modes we can choose between to *extended standby* and *power-save*. The difference between these two modes is that the system clock oscillator is stopped in *power-save* mode. This will give lower power consumption during sleep but increase the wake-up time because we have to wait for the oscillator to start up and the clock signal to be stable. The *extended standby* mode was chosen because we want the EPS to respond quick to interrupts. The power consumption will be a little bit higher compared to *power-save* mode, but this increase is negligible compared to the power consumption of other CubeSTAR subsystems.

3.5.3 Main Communication Bus

The I^2C bus has been chosen for both communication between modules and between the EPS and the various sensors. The reasons for choosing this bus are many. You only need two wires, clock and data, which makes PCB routing easier. You can achieve relatively high speed but we chose the lowest speed (100 kHz) because it is sufficient for our satellite. The I^2C bus has a seven bit addressing system. This makes a very flexible system suited for many nodes. It is bidirectional and supports multiple masters on the same bus. There is sufficient I^2C hardware and firmware support provided by the XMEGA manufacturer. All these features makes the I^2C bus suitable for the CubeSTAR mission.

The different modules regulate to different supply voltages. This means there are different I/O levels on the different modules. Some use 3.3 V, some use 2.5 V and some use unregulated power. To be able to disconnect a module from the communication bus, we need to use a I^2C bus buffer with disable functionality. For this purpose the LTC4301 from Linear Technology was chosen. The LTC4301 is supply independent, so it solves

the problem with different I/O levels. The pull-up resistors on the two sides of the buffer can be tied to different voltage levels. See figure 3.10 for the I^2C bus implementation.

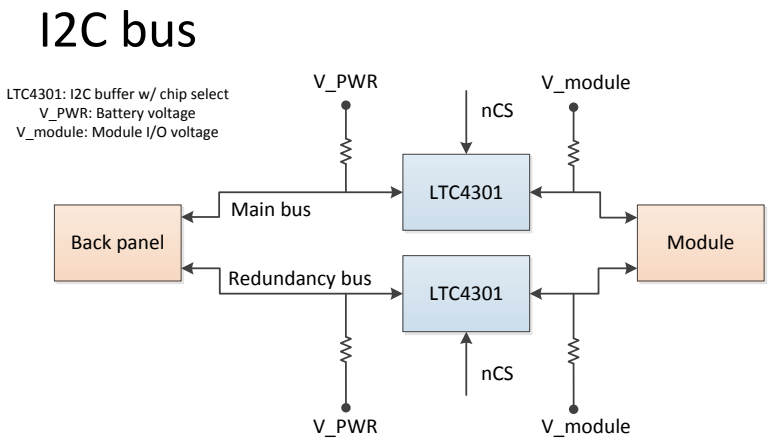


Figure 3.10: CubeSTAR communication bus block diagram.

To be able to power-toggle the modules, we need separate pull-up resistors on each side of the buffer. Otherwise we can actually power a low-powered device through a pull-up resistor. Virtually all CMOS devices are equipped with over-voltage protection, often implemented using clamping diodes. These diodes will clamp any over-voltage applied to a pin to Vcc+V_f and under-voltage to GND-V_f, where V_f is the forward voltage drop of the diode. If Vcc is set to 0 V and the pull-up is connected to 3.3 V, then the clamping diode will be forward biased and low powered devices like a MCU or sensor can actually be powered through this pull-up resistor, see figure 3.11. To avoid

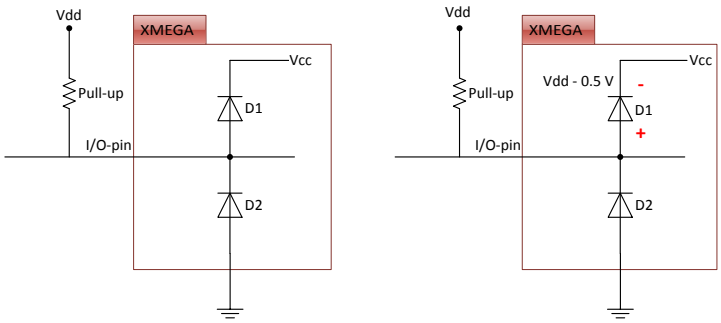


Figure 3.11: Powering a device through pull-up resistor and clamping diode. When power supply (Vcc) is set to 0 V, diode D1 is conducting.

this, we need the I^2C buffer to implement separate pull-up resistors on each side of the buffer. In addition, the buffer will help if we have problem with slow rise and fall times due to many devices connected to the bus or long I^2C wires.

3.6 Microcontroller Firmware

The firmware written for the MCU consists of mainly three tasks. First, it is a slave device on the main communication bus. It must react to commands issued by the master device, the OBC. Second, it is a master device on the EPS sensor bus. All the sensors in the EPS are connected to this bus and sensor data must be gathered and processed. This is called housekeeping and will be executed every 10th second. Third, it responds to faults on the power bus. The EPS firmware is based on interrupts. This is most efficient when it comes to power consumption. The MCU is woken up from sleep when an interrupt is received. When finished executing the interrupt service routine (ISR), the MCU resumes sleeping. See figure 3.12 for EPS firmware flowchart. For details about the firmware, see the source code in appendix E.

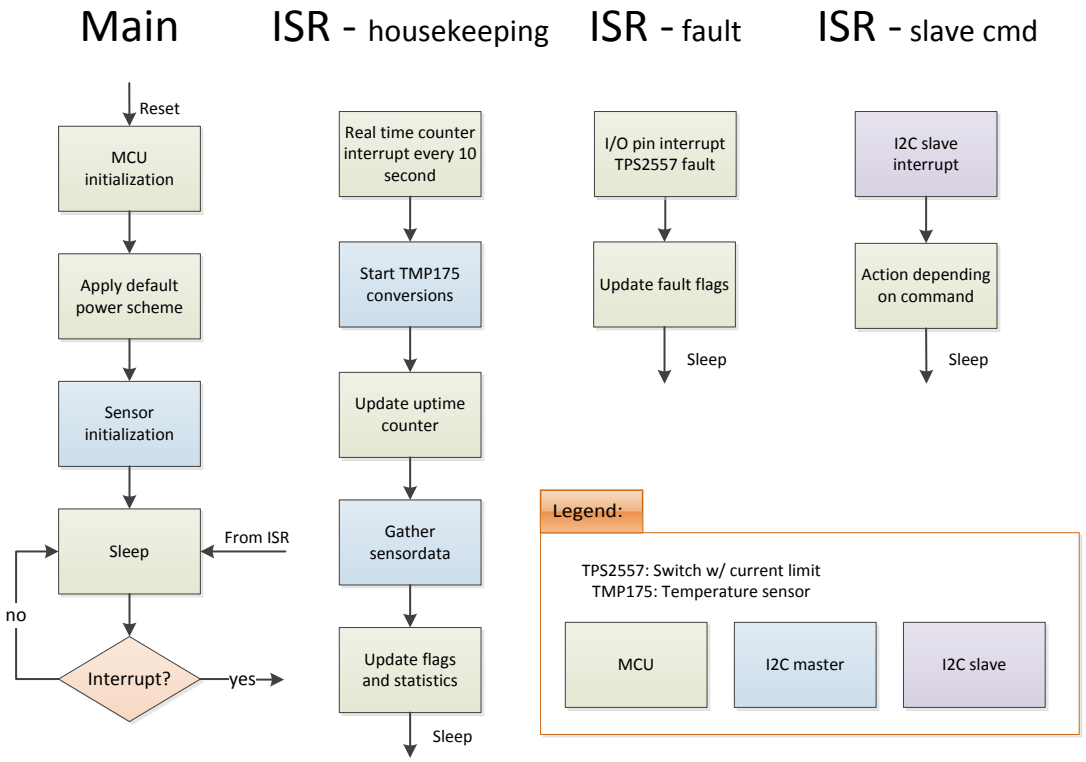


Figure 3.12: EPS firmware flowchart.

3.6.1 Firmware Modes

It has been implemented three different firmware modes that can easily be changed. The three different modes are `NORMAL` mode, `SENSOR` mode and `DEBUG` mode. The `NORMAL` mode is the mode intended for operation in orbit. The `SENSOR` mode is for sending all sensor data to the Universal Asynchronous Receiver/Transmitter (UART) bus. This is useful for sensor and integration testing. The various sensors can be tested and the power usage of the different modules can be measured. The `DEBUG` mode is the same as `NORMAL` mode but with printing system changes to UART. This is useful when testing the various commands from the OBC or alerts from sensors. The `DEBUG` and `SENSOR` can be combined to print both sensor data and debug messages to UART at the same time.

3.6.2 Housekeeping

The housekeeping task is interrupt driven and is triggered every 10 second by overflow in the RTC. First, the housekeeping task is starting the temperature sensor measurement because they need 27.5 ms to finish the conversion. Second, the uptime counter is updated. Third, the MCU starts gathering data from all the sensors and the data is processed before storing them. The EPS is doing simple statistics of the sensors. This means storing the maximum and minimum value in addition to the most recent sensor data. See figure 3.13 for the flowchart of the sensor data gathering and processing part of the housekeeping task.

To be able to interpret the sensor data, we need to know the format of the data and the scaling factor. Because the MCU has 8-bit architecture, we do not use floating point variables to store sensor data. 8 bit and 16 bit integers are only used. The INA226 is using a 16 bit ADC. From the datasheet of the INA226 we find that the scaling factor of the voltage measurement is 1.25 mV. The current measurement is done with a shunt resistor of 25 mΩ. The shunt voltage scaling factor is 2.5 μV. From Ohm's law, we can compute the shunt current scaling factor:

$$\begin{aligned} I &= \frac{2.5\,\mu\mathrm{V}}{25\,\mathrm{m}\Omega} \\ &= 0.1\,\mathrm{mA}. \end{aligned}$$

The INA226 is set up to do the conversion from voltage to current automatically.

The STC3100 is configured to use a shunt current resolution of 14 bit and a shunt resistor of 25 mΩ. We read the upper 16 bits of the 28 bit SoC accumulator. The scaling factor of the SoC value can be computed similarly to the INA226 current measurement:

$$\begin{aligned} SoC &= \frac{6.7\,\mu\mathrm{Vh}}{25\,\mathrm{m}\Omega} \\ &= 0.268\,\mathrm{mAh}. \end{aligned}$$

The TMP175 is configured to use 9 bit resolution. From the datasheet we find a scaling factor of 0.5°C. Because we have limited bandwidth on the satellite downlink, we

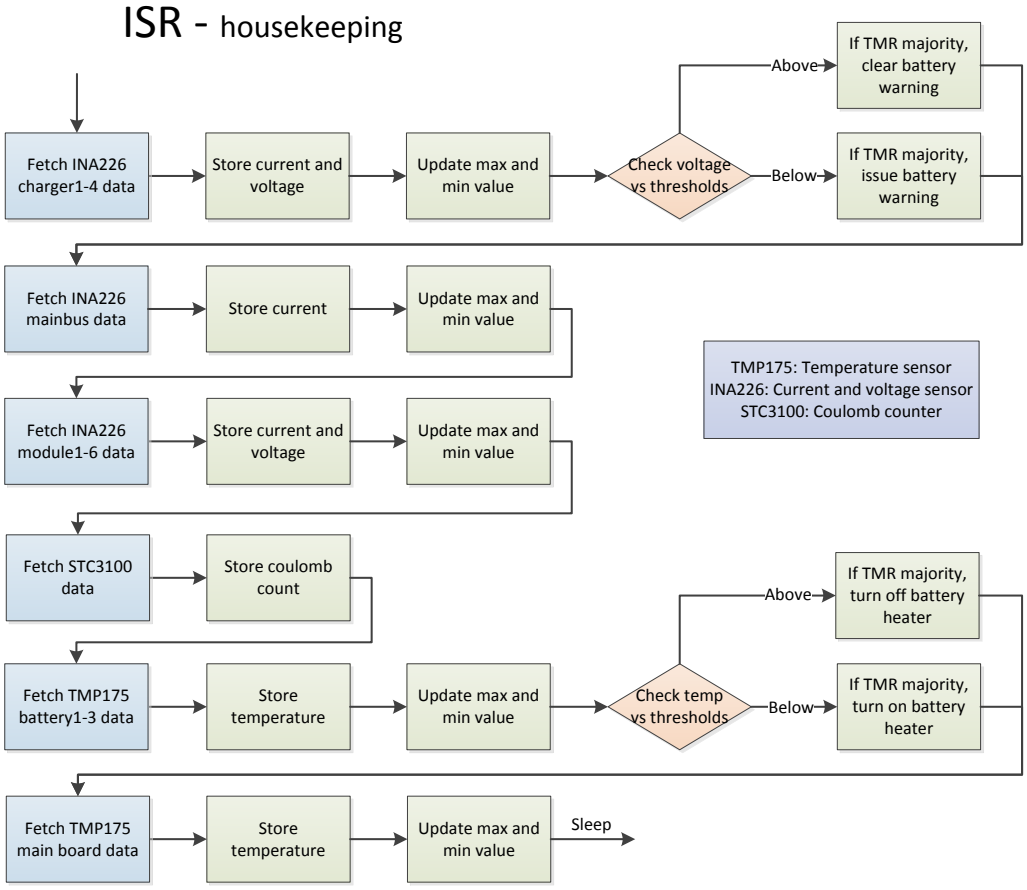


Figure 3.13: Sensor data gathering and processing flowchart.

do not want to use two bytes for 9 bits temperature data. The temperature readings are shifted right by one bit to reduce the resolution to 8 bit, giving a scaling factor of 1°C. All values are signed values in 2's complement form to be able to represent negative values. For more information about the sensors, see their respective datasheets. See table 3.5 for a summary of the processed sensor data properties.

Table 3.5: Processed sensor data properties.

Measurement	Data size	Scaling factor
INA226 voltage	16 bit	1.25 mV
INA226 current	16 bit	0.1 mA
STC3100 SoC	16 bit	0.268 mAh
TMP175 temperature	8 bit	1°C

Another important aspect of the sensor data properties is the endianess. XMEGA have little endian architecture. This means the least significant byte is stored at the lowest address. This must be taken into account when decoding the sensor data received at the ground station. See figure 3.14 for the difference between big and little endian on 8-bit processor architecture.

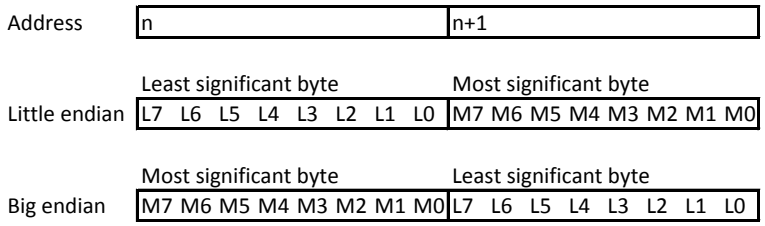


Figure 3.14: Endianess on 8-bit processor architecture.

3.6.3 Power Switch Fault

The TPS2557 issue an alert if the switch experience an over-current or over-temperature condition. These conditions were discussed earlier. The switch limits the current and after the deglitch time of 9 ms, it pulls the fault pin low thus disabling the switch and generating an interrupt on the MCU. The corresponding fault flag of the switch causing the interrupt will be raised. If the fault pin is set high again, the corresponding fault flag is cleared.

3.6.4 Slave Commands

The EPS is a slave unit on the main communication bus. The master, OBC, will send commands that the EPS has to respond properly to. See figure 3.15 for a flowchart of all the slave commands. See table 3.6 for a list of all the commands. We will briefly explain the different commands.

Status
The status byte consists of four flags, see table 3.7. The EPS_OFF flag should be true whenever any of the four modules are not powered. The EPS_BW should be true as long as the battery voltage is lower than recommended for normal operation. The EPS_NSE should be true when an event in the EPS has occurred. Every time a flag in the diagnostic flag is updated, the EPS_NSE flag will be raised. The EPS_USE should be true if any unexpected behaviour is detected. If the program reach an unknown or forbidden state, the EPS_USE flag is raised. The only alternative of correcting this error is to run the restart command of the EPS, thus setting the MCU to the default state. The EPS_NSE and EPS_USE flags should be cleared each time after sending the status byte to the OBC. The EPS_OFF and EPS_BW flag should be maintained as long as the condition raising the flag is present.

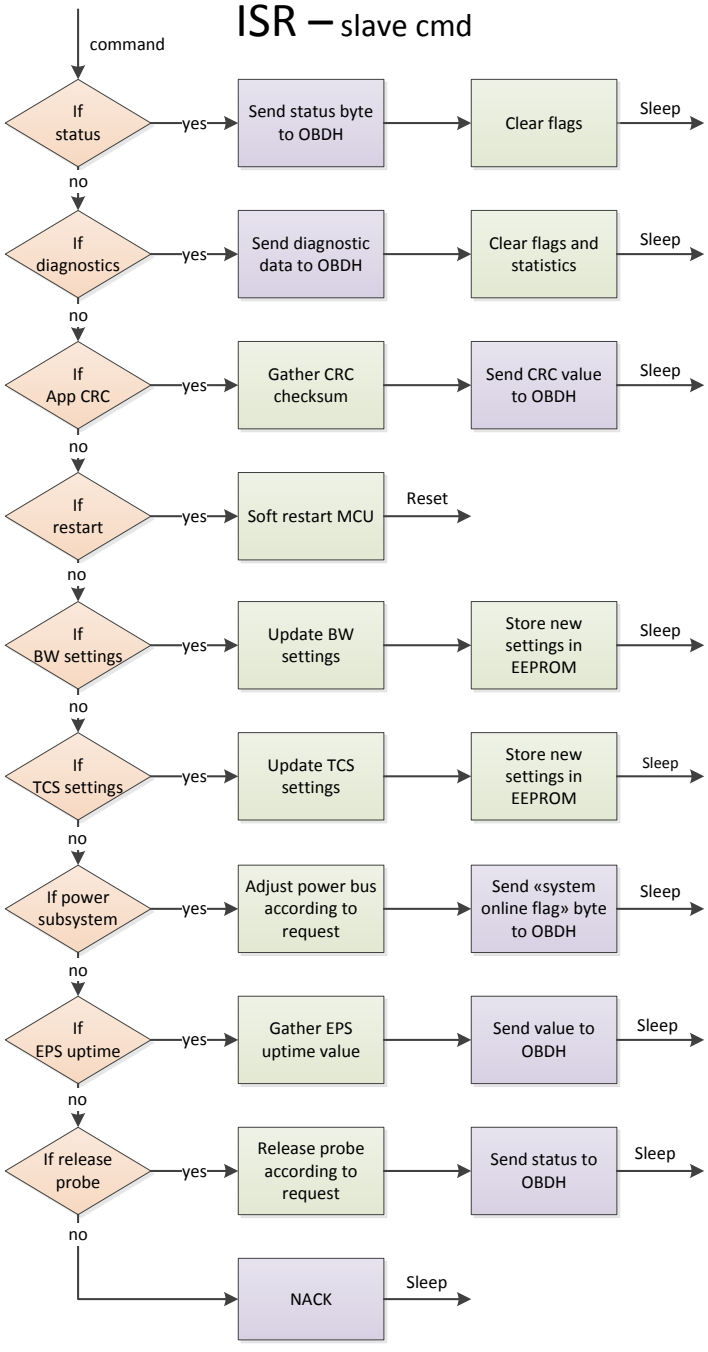


Figure 3.15: Slave commands flowchart.

Table 3.6: Slave commands.

Command	Description
Status	Return status byte
Diagnostics	Return diagnostic pack
Application CRC	Return 24 bit CRC
Restart	Reset MCU
BW settings	Adjust BW thresholds and store in EEPROM
TCS settings	Adjust TCS thresholds and store in EEPROM
Power subsystem	Adjust power distribution and return status
Uptime	Return uptime since last start-up
Release m-NLP	Enable release circuit and return status
CC reset	Coulomb counter charge register reset

Table 3.7: Status byte.

Flag	Description
EPS_OFF	True if any subsystem is offline
EPS_BW	True if battery warning
EPS_NSE	True if any new system event
EPS_USE	True if any unhandled system error

Diagnostic
The diagnostic pack consists of several flags and all the processed sensor values. Each time the EPS_NSE flag is set, the OBC will gather the diagnostic pack from the EPS. See table 3.8 for an overview of the diagnostic pack. For more details about the contents in the diagnostic pack, see the source code in appendix E.

Application CRC
The application Cyclic Redundancy Check (CRC) can tell us whether the application code on the MCU is corrupted or not. The CRC value gathered from the EPS in orbit should be identical to the CRC value generated before launch.

Restart
The restart command will initiate a reset of the MCU. This reset will set the MCU to the default state and the program counter will jump to address 0.

BW Settings
This command updates the set and clear threshold for the battery warning. The new values are stored in EEPROM. At startup the EPS checks the content of the EEPROM. If the EEPROM is empty (0xFF), the default BW settings are loaded. If the EPS

Table 3.8: Diagnostic pack.

Flag	Description
packageID	Fixed value 0x2D
epsID	EPS slave address 0x0E
sysEventFlag	EPS event ID
restartFlag	EPS reset flags
sysOnline	System online flags
enable_1_4	TPS2557 enable flags
enable_5_8	TPS2557 enable flags
fault_1_4	TPS2557 fault flags
enable_9_10	TPS2557 enable and fault flags
i2c_transaction_result	Store result if I^2C error
i2c_bus_state	Store state if I^2C error
sensor_fault_1	Store ID of faulty sensor
sensor_fault_2	Store ID of faulty sensor
sensor data	All the sensor data and statistics
releaseStatus	m-NLP release status flags
BW_set	Battery warning set threshold
BW_clear	Battery warning clear threshold
TCS_state	TCS on/off flag
TCS_enable	TCS enable threshold
TCS_disable	TCS disable threshold
uptime_ticks	Uptime 10 second tick counter
uptime_weeks	Uptime week counter

find any BW settings in the EEPROM, these settings are loaded instead of the default settings.

TCS Settings
This command updates the global on/off flag and enable/disable thresholds of the TCS. The values are stored and loaded from EEPROM in the same way like the BW settings.

Power Subsystem
The power subsystem command will tell the EPS to either power on, power off or power-toggle modules. In normal operation all modules should be powered on, but if a battery warning is detected the OBC will start to disable all modules except the EPS and OBC. If a module does not respond as expected, a power-toggle can be tried to correct the error. The power-toggle command need to implement a delay before turning the power back on again. This because the capacitor on the output of the power distribution switch need some time to discharge. The response byte consists of the power status of the four modules.

Uptime
The uptime command tells the EPS to return the value of the uptime counter. Every 10th second, when the RTC will overflow, the uptime counter is updated. This counter tells us the uptime of the EPS since last start-up.

Release m-NLP
The release m-NLP command will tell the EPS to enable power to a m-NLP release circuit. Because the current drawn is relatively high, around 500 mA, only one probe is released at a time. The response byte consists of the release status of all four probes.

Coulomb Counter Reset
This command will reset the coulomb charge accumulator counter. We discussed earlier that an offset can build up over weeks. This command can be used if you want to remove this offset from the coulomb counter.

3.7 Fault Tolerance and Error Handling

3.7.1 Passive Components

To increase the fault tolerance, redundancy is introduced in both the kill switch and the RBF switch. Both the RBF switch and kill switch are implemented as two switches in parallel. If one switch starts malfunctioning, the other switch is enough to maintain normal operation. Two separate deployment switches are implemented. If one of the spring-loaded pins get stuck during launch, the satellite will still operate as normal using only one of the kill switches.

The shunt resistors of the various current sensors are inserted in series with the power bus. To increase the fault tolerance, two resistors in parallel has been implemented. In case one of the resistors stop conducting, the current reading will be doubled due to the doubled resistance. This should be fairly easy to detect and corrected by the ground station crew.

3.7.2 Active Components

CubeSTAR is designed using COTS components and none of the ICs in use are space qualified. Because we do not have any radiation tolerance characteristics, we do not know if SEUs or SELs can be a problem. Components like the SPV1040 and TPS2557 has no register storage as far as we know. This means that these devices are probably not vulnerable to SEUs but SELs can be a problem. Both the SPV1040 and TPS2557 are over-temperature protected. A SEL will cause a short-circuit current somewhere inside the device which will result in a temperature increase. We rely on the over-temperature protection to shut the device down, thus removing the SEL before any permanent damage occurs.

The sensors like INA226 and TMP175 can be vulnerable to both SEUs and SELs. The sensors are protected from SELs by the auto-retry functionality on the sensor power bus.

Most of the sensor data will only be sent down to the ground station for informational purposes, thus it is not critical if some of the sensor readings become corrupted by SEUs. Critical sensor readings, like low battery voltage and low battery temperature, will be protected by TMR. There are implemented three separate sensors for both the battery voltage and battery temperature measurement. The voting system is implemented on the EPS MCU. The MCU accepts the low-voltage or low-temperature warning if the majority of the sensor measurements, at least two out of three, are concurring.

3.7.3 Microcontroller

The MCU is vulnerable to SEUs. To handle this, Myrland has implemented Hamming error correcting codes [14]. Both the commands issued by the OBC and the EPS status and diagnostic flags are protected by this error correcting scheme.

In case of an EPS processor failure, the satellite must go into a default state. In the default state only basic functionality is enabled. We have decided that the default state is OBC and COMM enabled. This is accomplished by tying the enable signals of the various module power switches to a default level using external pull-up and pull-down resistors.

The battery warning is issued at 2.8 V battery voltage. To increase the fault tolerance, the I^2C buffers are powered directly from the battery not through the 3.3 V regulator. The minimum recommended supply voltage of the I^2C buffer is 2.7 V. If the I^2C communication between EPS and OBC stops working correctly during a battery warning, a redundant system has been implemented. There is a I/O line called EPS_IRQ connected directly between the EPS and OBC processors. If the EPS issue a battery warning, the EPS_IRQ is set high and kept high as long as the battery warning condition exists.

3.7.4 Firmware

The EPS firmware is implemented using a watchdog timer. If the EPS MCU gets stuck somewhere in the firmware, a watchdog timer will overflow and cause a reset of the EPS. After a reset, the MCU is re-initialized thus rewriting the SRAM cells that are possibly corrupted by SEUs. If the EPS reach a forbidden or unknown state, this is reported to the OBC using the EPS_USE flag. The OBC will likely try a restart command.

If one sensor does not respond, the sensor value is set to a non-valid value. The flag which indicate *sensors not responding* is set and the identification of the faulty sensor is logged. The non-valid value is defined to be the maximum negative value which is *0x8000* for 16 bit data and *0x80* for 8 bit data. If we get a sensor reading that is outside the predefined allowed range, it means the sensor reading is faulty. Then the flag which indicates sensor readings outside the allowed range is set and the identification of the sensor is logged. In both cases the statistics is not updated and the EPS will issue a power-toggle of all sensors at the end of the housekeeping task. This will hopefully correct the error, otherwise there are no options of correcting sensor errors.

Chapter 4

Tests and Results

4.1 SPV1040 Test

From the SPV1040 datasheet we can expect up to maximum 91-92% efficiency. This test was carried out using a single-layer PCB with one SPV1040 charger and two current monitors. The current monitor chosen for this test was the Texas Instruments INA214. INA214 has analog outputs, otherwise it is similar to INA226. INA226 was not used because at the time of testing the data logging facilities with I^2C interface was not operational yet. Voltage and current from the PV-cells and voltage and current from the charger into the battery, was instead logged using a Labview data acquisition program. Please see appendix C for PCB schematics, layout and the Labview program.

4.1.1 Test Setup

The data acquisition device (NI USB-6008) was used in Referenced Single-ended (RSE) mode. Some of the measurement noise can be filtered out during post-processing. The total noise level using RSE mode was found to be acceptable for making a conclusion of these tests. These tests were carried out using two PV-cells in parallel into a SPV1040 and a single battery cell on the output. The weather conditions were sunlight from a clear sky except if noted otherwise.

Test equipment:

- National Instruments USB-6008
- Fluke 75 multimeter
- Spectrolab 28.3% UTJ PV-cell
- A123 APR18650M1A $LiFePO_4$ single cell 1.1 Ah battery
- Tektronix TDS 1002 oscilloscope

4.1.2 Maximum Power Point Tracking

A V_{oc} of 2.45 V on the PV-cell was measured with the TDS 1002 oscilloscope. From the datasheet we can find an V_{oc} of 2.66 V and the difference is probably due to temperature difference. The numbers in the datasheet is for 28°C. The temperature was lower when the V_{oc} was measured but the actual temperature was not recorded. The PV-cells used was cheaper 2nd grade cells that have a lower specification compared to the datasheet, which could explain the difference. As we saw in chapter 2, the MPPT algorithm adjusts the loading of the PV-cell until reaching the maximum power point. This means that the voltage on the PV-cell will be constantly changing. In figure 4.1 we can see the voltage on the PV-cell when the SPV1040 is connected. As we can see the voltage is oscillating. From what we discussed in chapter 2, this is evidence that the MPPT algorithm is working. The charger is adjusting the load up and down to always stay around the maximum power point.

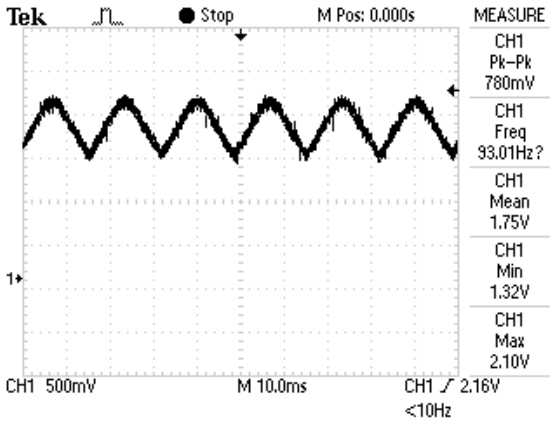
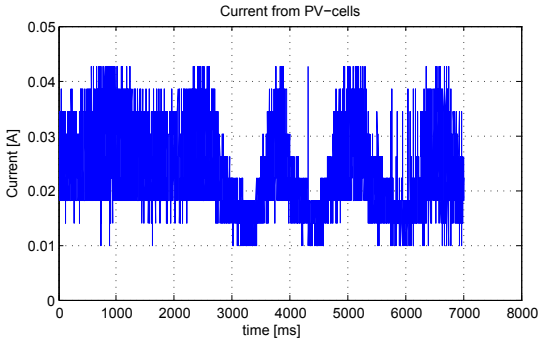


Figure 4.1: Voltage on PV-cell output when connected to the charger SPV1040. The MPPT algorithm is causing the voltage oscillation.

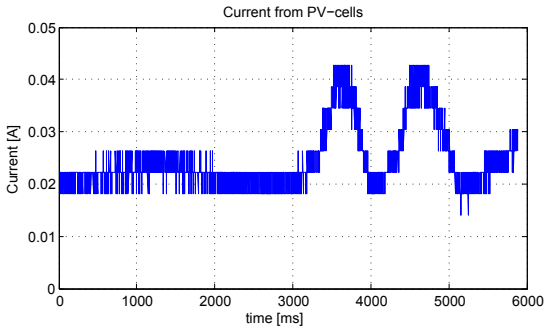
4.1.3 Current Sensor Noise

As discussed in chapter 3, a passive LP-filter on the current sensor inputs has been implemented. We can see the result of the filtering in figure 4.2[1]. We can see that the passive LP-filter is helping out in filtering high frequency noise. Post processing of the data in Matlab was considered to get rid of some of the random noise and to smooth the variations introduced by the MPPT algorithm. When measuring the currents and voltages with a multimeter, a stable DC-value is found. This means that the multimeter is doing some filtering to be able to display a stable value. Both an arithmetic mean

[1]The reason for the low current is that this measurement was done with a 40 Watt light bulb, not sunlight.



(4.2a) Filtering off.



(4.2b) Filtering on.

Figure 4.2: Showing the effect of passive LP-filter with cutoff frequency at 11.7 kHz.

(4.1) and a moving average (4.2) filter was implemented to get a more stable signal.

$$y_{am} = \frac{1}{N} \sum_{i=0}^{N-1} x_i \tag{4.1}$$

$$y_{ma} = \frac{x_i + x_{i-1} + ... + x_{i-(N-1)}}{N} \tag{4.2}$$

Arithmetic mean is a good filter if you can oversample the signal because you decimate (down-sample) the signal when you do the arithmetic mean. It is very good for suppressing random noise. If you know the period of periodic noise, you can adjust the length of the arithmetic mean to reduce the periodic component, e.g. 50 Hz noise from the AC power lines. Moving average filter is a good filter for real-time processing because you only need present and previous samples to compute the result, not future samples. The moving average filter is good for smoothing out a varying signal [2]. Averaging over 100 samples and a moving average filter of length 3, has given satisfactory result for our tests. Plots showing the effect of the filtering is found in appendix D.

4.1.4 Charger Efficiency

The Labview DAQ program is logging the voltage (V_{in}) and current (I_{in}) from the PV-cells and the charger current (I_{out}) and battery voltage (V_{out}). From this we can compute the input and output power of the charger. The charger efficiency (η) is defined as the ratio between the output power and the input power:

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out}I_{out}}{V_{in}I_{in}}.$$

After smoothing and averaging the data a charger efficiency of around 83% was found. When measuring the efficiency with multimeters the result was 83.6%. These results agree well with each other. Later on some of the passive components around the charger was changed. More specifically, the transient voltage suppressor diode was changed to another with smaller leakage current and the capacitor on the charger input and output was changed from tantalum to ceramic. This is recommended in the charger datasheet and the result was an increase in efficiency to 85%. See appendix D for plots of the efficiency tests.

4.1.5 Angle of Incidence

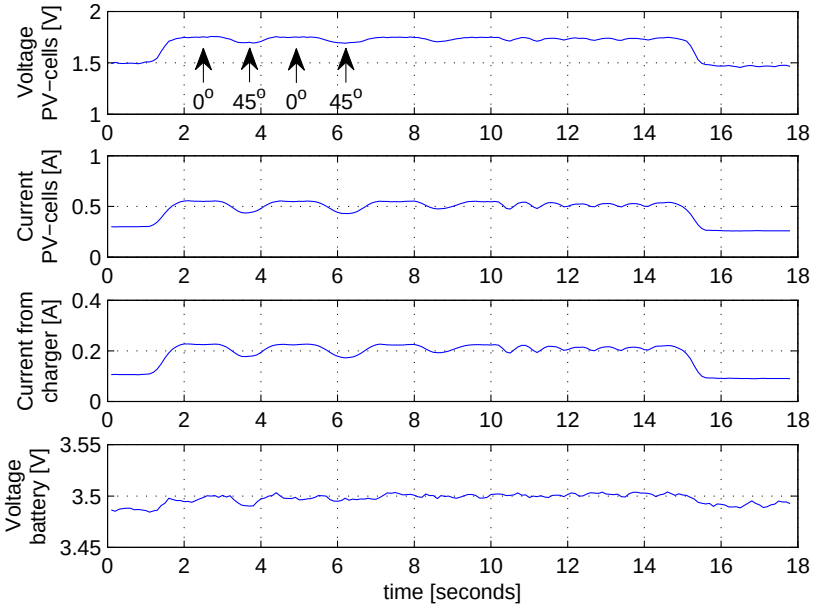
Depending on how well the ADCS works, the satellite will rotate during orbit. We cannot expect to have a fixed angle towards the Sun while orbiting. Therefore, it is important to see how quickly the charger reacts to changes in the angle of incidence. In figure 4.3 we see that the charger output follows the input almost instantaneous. We do not know how fast the satellite will be rotating but hopefully not more than a few hertz. The angle of incidence changes were done by hand force and we can see that the charger reacts quickly to changes up to around 1 Hz. We can see that 45° angle of incidence is only lowering the PV-cell voltage by approx. 50 mV. When the angle of incidence is 90° the PV-cell voltage is lowered by approx. 500 mV.

4.2 Battery Temperature Test

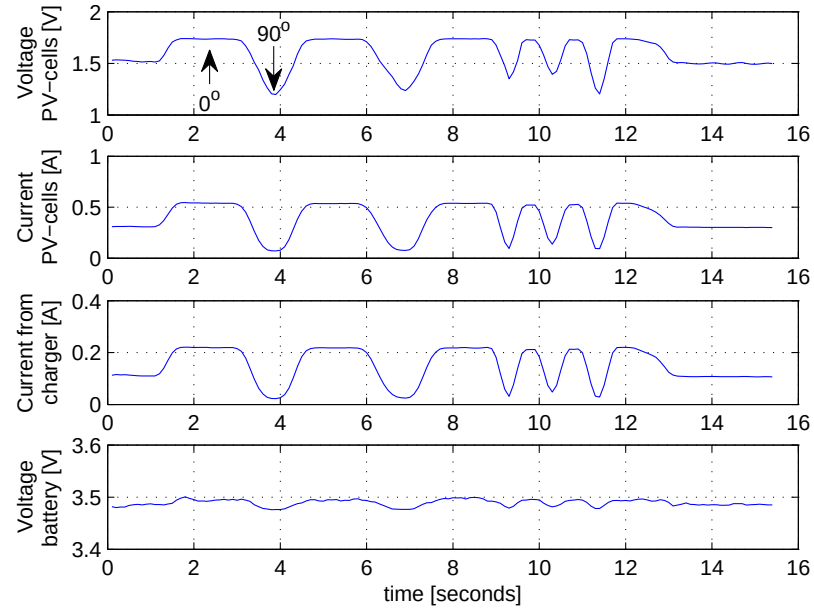
Temperature is an important parameter when it comes to battery performance, both battery voltage and usable capacity. Our goal is to keep the battery voltage above 2.8 V for the greater part of the discharge curve. From the battery datasheet we find the discharge characteristics at 25°, 0° and -20°C. We find that somewhere between 0° and -20°C the battery voltage quickly becomes too low for our application. It is important to find out at what temperature the voltage becomes too low, because this will tell us when the battery heating system should be activated. The battery was characterized when exposed to temperatures of 23°, 0°, -5°, -10°, -15° and -20°C.

4.2.1 Test Setup

To reduce the charge/discharge time, a single battery cell was used in this test. The battery was charged to 95% with a TP610C charger and put the back panel and battery



(4.3a) Incident angle changing between 0 - 45°.



(4.3b) Incident angle changing between 0 - 90°.

Figure 4.3: Changing angle of incidence by hand force. We see how the angle of incidence affects the PV-cell voltage and current. We can see that the SPV1040 charger is quickly adjusting to changes in input power.

module into the test chamber. A temperature test chamber from Associated Testing Laboratories Inc. was used. This test chamber is old and has problems of maintaining stable temperature, especially at low temperatures. Slow variations of 1° to 2°C around our wanted test temperature was experienced. This variance is acceptable, because it is not critical to find the exact temperature range for the battery heater operation. The possibility of changing the TCS enable/disable threshold while deployed in space has been implemented. If the optimum thresholds is missed, the thresholds could be adjusted later. In addition, it is interesting to see how the battery reacts to these small temperature variations.

The temperature was stabilized and a discharge load of 3.3 Ω was connected. This results in a discharge current of approx. 1 A, at least before the battery voltage starts to decrease. The reason for choosing this load is that 1 A is the highest expected load current on the satellite, which occurs during radio transmission. The battery voltage and temperature was logged during discharge and the battery was discharged until reaching 2.5 V. INA226 was used for the voltage measurement and the TMP175 located on the battery module was used to log the temperatures. All temperature plots are found in appendix D. We remind you that the TMP175 sensor data has a typical accuracy of ±0.5°C.

4.2.2 Test Results

In figure 4.4 we see the battery discharge characteristics for all the temperatures measured. We see that the lower the temperature gets, the battery voltage and capacity is increasingly dependent on the temperature. For -22°C and -15°C we can see the small temperature changes in the test chamber directly affecting the battery voltage up to 50 mV. From the figure it seems like the battery capacity is lower at 23°C compared to 0°C but this is due to higher battery voltage at 23°C resulting in a larger discharge current.

These measurements are worst-case measurements, because they were done with the heaviest expected load current. During normal operation, this load current will only occur in short transients. It is important that the satellite can withstand the worst-case current consumption without temporary lowering the battery voltage below the battery warning threshold.

The battery pack is oversized, which means the battery pack will only be discharged 5-10% during normal operation. Thus it is most interesting to see the first 5-10 minutes of the discharge curve, but in case the extra battery capacity is suddenly needed, we need to assure high enough battery voltage for the greater part of the discharge curve.

From these results we can see that down to -5°C, we have a battery voltage well above 2.8 V for most part of the discharge curve. When reaching -10°C, the battery voltage starts to decrease below 3.0 V significantly earlier. This tell us that the battery temperature should be kept above -10°C. It is suggested that the TCS is enabled at -7°C and disabled at -5°C. This should ensure battery voltage above 2.8 V for most part of the discharge curve.

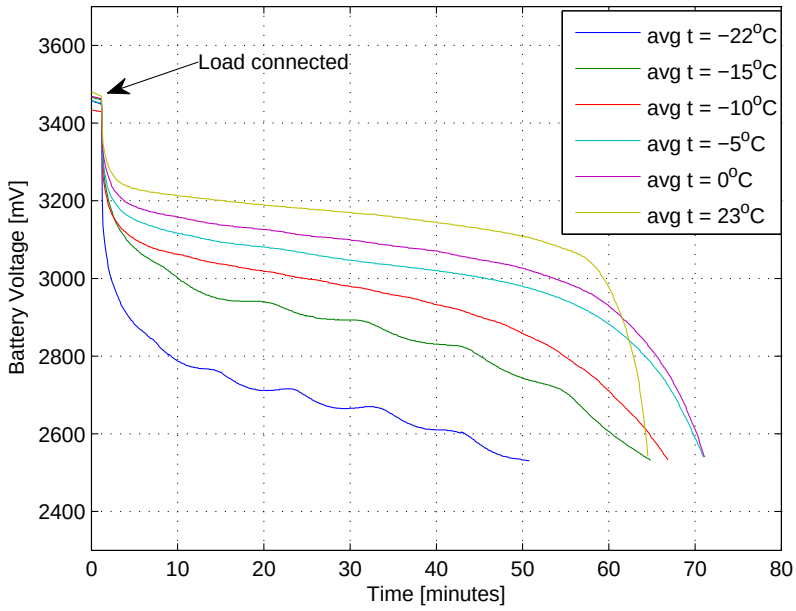


Figure 4.4: Battery discharge characteristics vs. temperature. Battery discharged into 3.3 Ω, making the discharge current dependent on the voltage. The temperature was slowly changing 1° to 2° around the stated temperature.

4.3 PCB Realization

In this section we will take a look at the realization of the PCBs and discuss some of the challenges experienced.

4.3.1 Battery Pack

In figure 4.5 we can see a picture of the battery module with one battery cell connected. All the components are fitted between the battery cells. We see three TMP175 temperature sensors, a STC3100 coulomb counter and an INA226 current sensor. The TCS is fitted between the battery cells as well, two heating resistors on each side of the PCB. All modules have a male connector on the module side. The battery pack must not be short circuited because each battery cell can give 30 A continuous current. It was chosen to reverse the polarity of the connectors on the battery pack. We see that the battery pack connector is female, thus reducing the risk of accidently short circuiting the battery pack.

4.3.2 Back Panel

In figure 4.6 we can see the second version of the back panel PCB. We see four identical charger circuits and the module slots which includes I^2C buffers, power distribution and

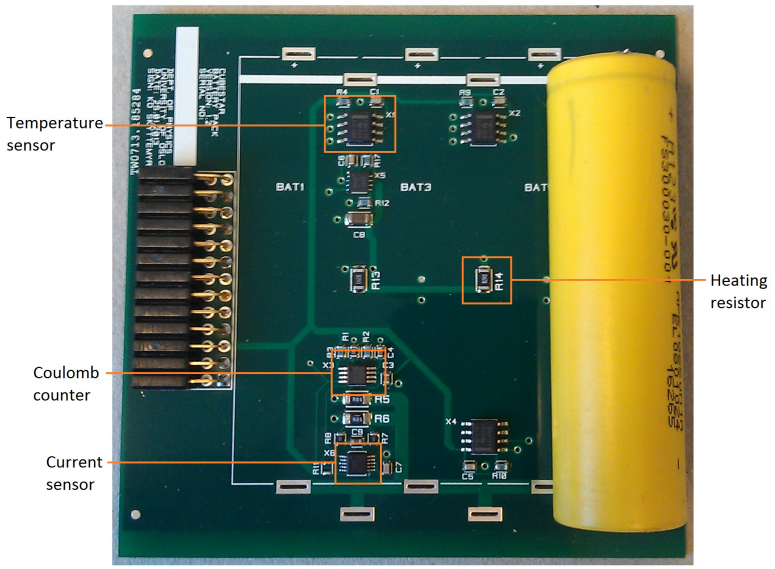


Figure 4.5: Battery pack PCB realization with one battery cell mounted.

current/voltage monitors. We see the microcontroller, the 3.3 V regulator, and the kill switch and RBF pin.

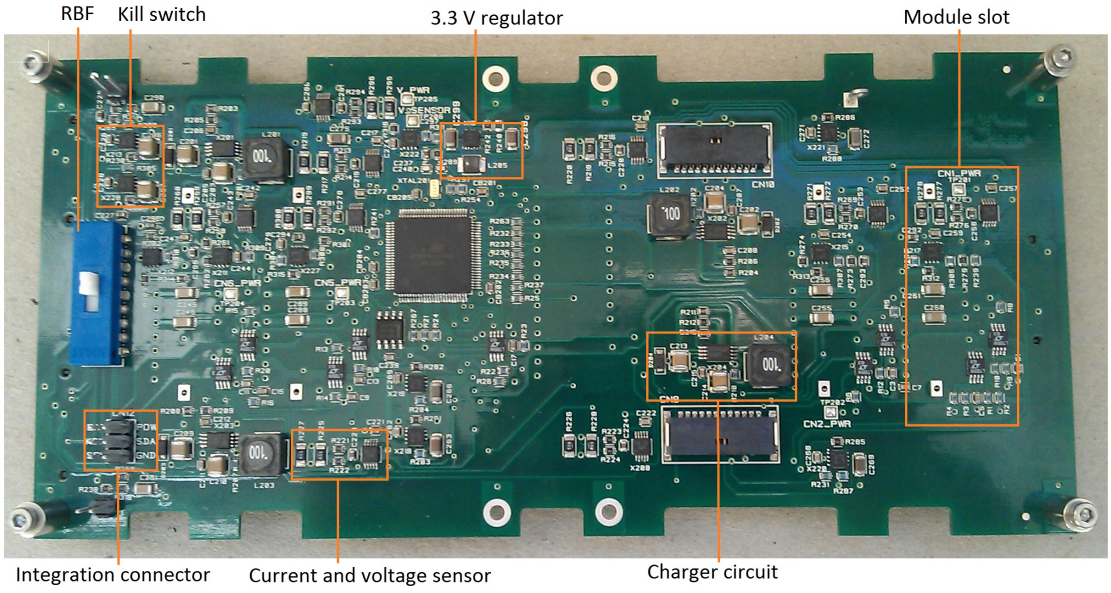


Figure 4.6: The second version of the back panel PCB bottom side.

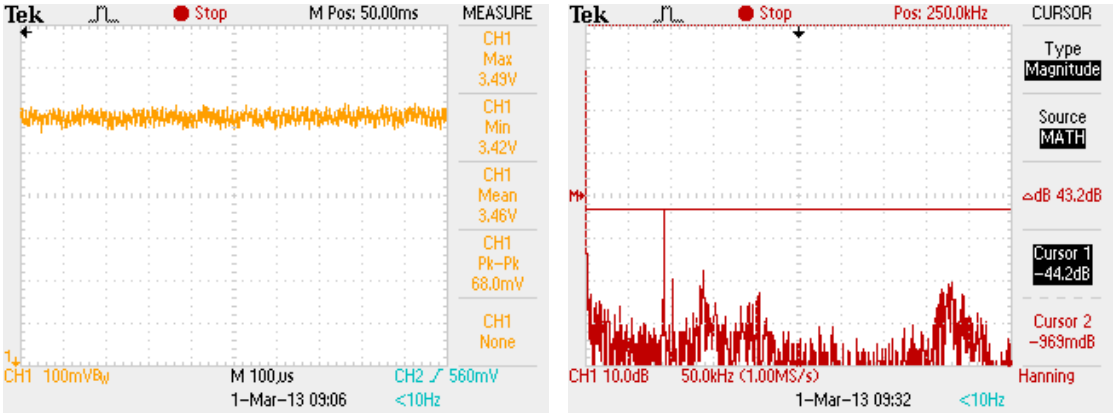
A charger efficiency of 85% was measured on the SPV1040 test board. After that test an inductor with lower series resistance was found and the charger datasheet tell us that a 4 layer PCB with solid ground layer can help increasing the efficiency. The

charger efficiency was increased to 90% after implementing these improvements. This is very close to the 91-92% efficiency the manufacturer promise.

The voltage drop between the battery and the different module connectors was measured when drawing 100 mA out of the module connector. A voltage drop of 9 mV was measured, which equals a total power bus resistance of approx. 0.09 Ω. The main contributor to this resistance is the two current sensor resistances and the two TPS2557 MOSFET channel resistances. From the datasheets these components should add 83 mΩ, the rest of the resistance is from the PCB tracks themselves, the Remove Before Flight pin etc.

The EPS is always enabled during orbit, so the total power consumption of the EPS should be as low as possible. A current consumption of 20 mA on the EPS in *NORMAL* mode was measured. This includes all sensors, MCU, active components and various losses around on the battery pack and back panel PCB. 20 mA equals 66 mW at the nominal voltage of 3.3 V.

When using switched-mode regulators we must assume there will be some high frequency noise on the output. In figure 4.7 we see a measurement of the voltage on a module connector with a SPV1040 charger running. In 4.7a we see that the voltage ripple is 68 mV peak to peak. In 4.7b we see the result of a FFT of the voltage. The SPV1040 has a fundamental frequency of typically 100 kHz. In the measurement we can find this fundamental frequency around 90 kHz and with an amplitude of -44 dB referred to 1 V_{rms}. This equals an amplitude of 6.3 mV_{rms}.



(4.7a) Time domain *(4.7b) Frequency domain*

Figure 4.7: Power bus noise with SPV1040 charger running. We can find the fundamental switching frequency component of the SPV1040 charger around 90 kHz.

The auto-retry functionality of TPS2557 is an important part of the system, providing SEL removal and limiting the wasted power during a short-circuit. To test the auto-retry a 1.5 Ω resistor was connected to the output of a TPS2557 which triggered the over-current protection. In figure 4.8 we can see a plot of the enable signal and output voltage of the TPS2557. The output voltage (channel 2) is current limited during the deglitch time of 9 ms. The current limiter is lowering the output voltage from 3.3 V to

1 V. After the deglitch time of 9 ms, the switch opens and the output voltage quickly falls to 0 V. The retry capacitor starts to charge and the enable signal increases towards the enable threshold of TPS2557 which is 0.9 V. After a retry time of approx. 350 ms, the enable signal reaches 0.9 V and the switch is re-enabled.

Figure 4.8: Test of TPS2557 auto-retry functionality. Channel 1 (yellow) show the enable signal. Channel 2 (blue) show the output voltage.

The fault pin sinks current from the retry capacitor which has a maximum voltage of 3.6 V. The fault pin is an open-drain pin which can maximally sink 25 mA continuous current. This means we need a current limiting resistor on this pin. A 1 kΩ resistor was implemented but this is a too large value causing a significant voltage drop. Instead of pulling the enable signal down to 0 V, the fault pin was only able to pull down the enable signal to 450 mV. Information obtained from Texas Instruments tell us that the fault pin should be able to sink 50 mA peak current. This results in a resistor value of $\frac{3.6V}{50mA} = 72\Omega$. To be on the safe side, a resistor value of 100 Ω was chosen. Now the fault pin is able to pull down the enable signal to 200 mV which is an acceptable level.

4.4 Vacuum Chamber Tests

The EPS was tested in a vacuum chamber with a 2SYF-1B vacuum pump able to reduce the pressure to 0.5 Pa (5 μbar). This pressure is equal to an altitude of approx. 80 km while CubeSTAR will be launched into an altitude around 500 km. Even though there is a difference between the vacuum chamber and the expected pressure in orbit, this test will give us a good indication whether the chosen components can withstand the low pressure or not. From the sensor readings and debug messages there has not been found any evidence of malfunction while the EPS was running inside the vacuum chamber. Visual inspection of the PCBs has not revealed any buckling or cracking components.

To test the battery heating system the vacuum chamber was put into the same temperature chamber used to characterize the batteries. The temperature chamber was set to -10°C and the temperature stabilized. When the battery pack temperature sensors

showed -7°C, the vacuum pump was switched on. After removing the air the battery heater was enabled. In figure 4.9 we see that the 0.5 W heating system was able to raise the temperature on a single cell battery module by 5°C. The back panel and battery PCBs are thermally connected, so the temperature on the back panel was also increased a little bit. To increase the temperature by more than 5°C, the power of the heater could be increased. We remind you that the temperature data is shifted right to give a resolution of 1°C, e.g. 1.9° is rounded down to 1°.

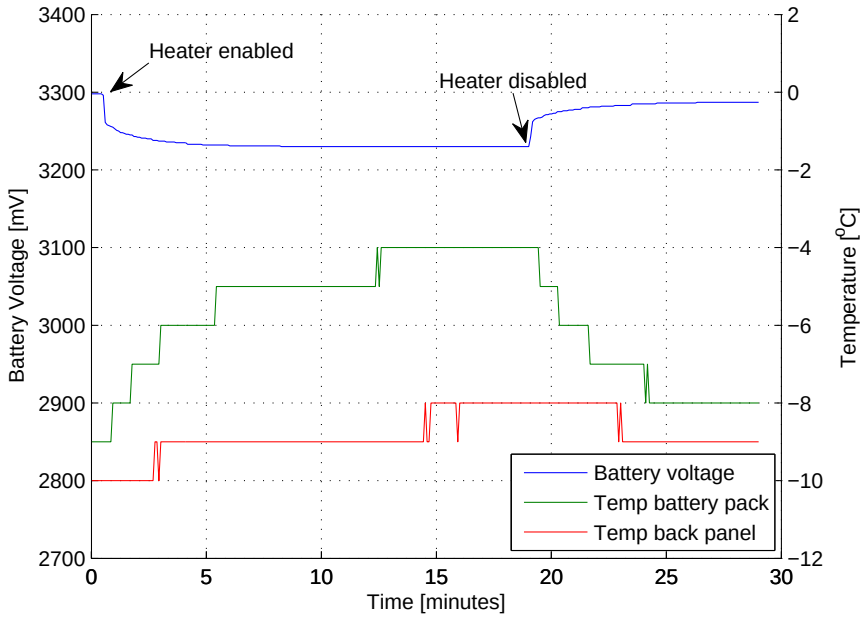


Figure 4.9: Test of 0.5 W battery heater in vacuum chamber with one battery cell mounted. The battery heater is able to raise the temperature 5° C. The temperature outside the vacuum chamber was -10° C.

In figure 4.10 we see the same test but with all six battery cells mounted on the battery module. We see that now is the battery heater only able to increase the temperature 2°C. The reason for this is that the battery cells increase the thermal inertia of the system because of the increased mass on the battery module. This means the battery heater will need more power to increase the temperature, but the thermal inertia works the same way both when going from hot to cold and cold to hot. This means the six cell battery module temperature will decrease slower when exposed to cold and this is exactly what we want: A stable battery module temperature.

Another test was performed to take a closer look at the thermal inertia of the EPS. The back panel and battery module is located inside the satellite. The side panels will give some isolation from thermal radiation into outer space. We have earlier seen that the temperature on the side panels can be in the range from -25° to 25°C. To simulate these conditions the temperature chamber was set to -20°C and the temperature inside

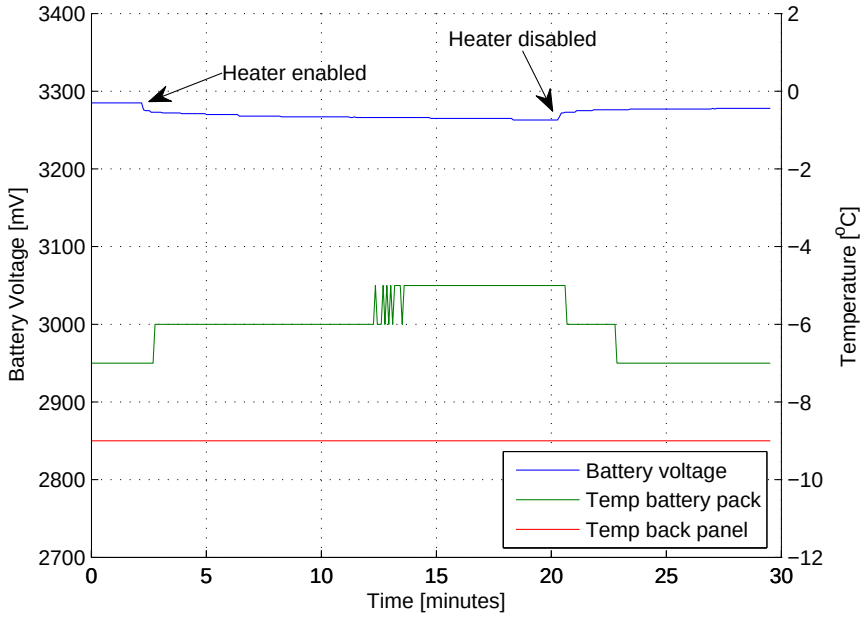


Figure 4.10: The same test as in figure 4.9 but with six battery cells mounted. The heater is now only able to raise the temperature 2° C.

the vacuum chamber was stabilized to 15°C. Then the vacuum chamber was put into the temperature chamber to see how quickly the temperature was falling. After 45 minutes the vacuum chamber was taken out into room temperature (18°C) to see how the temperature was rising. As we can see from figure 4.11 the thermal inertia of the battery pack is much larger than the back panel. In 45 minutes the temperature on the back panel dropped 27°C while the temperature on the battery pack dropped 17°C. We assume the eclipse is lasting around 30 minutes. This tell us that the battery heater is probably not needed because the temperature of the battery pack will be slowly changing. It has been decided to leave the battery heater default off and the heater can be enabled by the ground station crew if needed. We can see that the temperature on the battery pack is decreasing a little bit after the vacuum chamber is moved into room temperature. This reason might be that the battery cells had a slightly lower temperature than the temperature sensors. The temperature sensors are located between the PCB and the battery cells and it probably took some extra time for the temperature to stabilize between the battery cells and temperature sensors.

This test was performed with the back panel and battery module. The picture will probably look a bit different when all of the satellite is assembled. The back panel and all the subsystems, except the battery pack, is directly connected thermally to the aluminium structure. The battery pack is only thermally connected to the back panel.

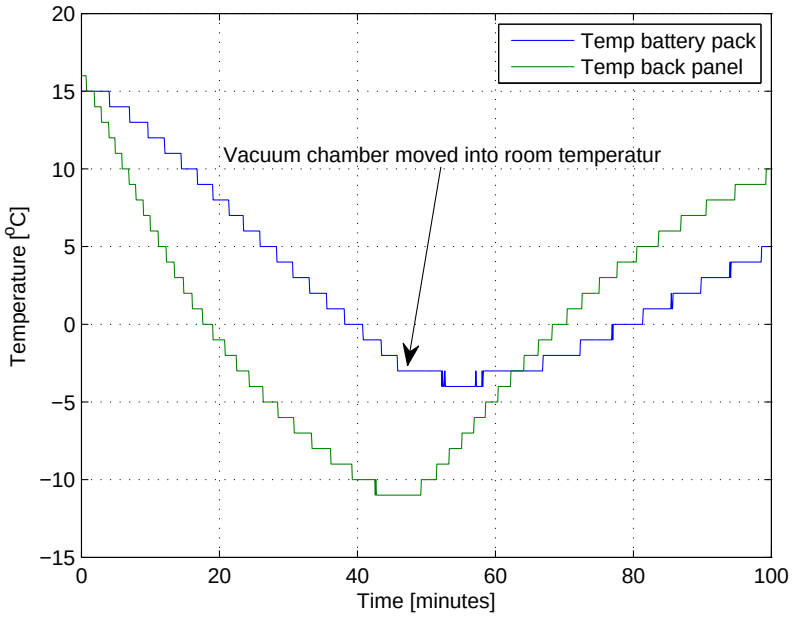


Figure 4.11: Test of thermal inertia of the EPS. The vacuum chamber was put into the temperature chamber (-20° C). After 45 minutes the vacuum chamber was put back into room temperature (18° C).

4.5 Sensor Accuracy

In chapter 3 the typical error of the EPS sensors was computed. A Fluke 289 has been used to check whether the sensor measurements are within the expected accuracy range.

4.5.1 Kelvin Connection

When testing the first version of the back panel, a rather larger error on the current measurements was found. Kelvin connection is important when measuring current using a small shunt resistor [19]. If we do not connect the PCB traces from the current sensor exactly on the shunt resistor pads, the effective resistance in the shunt will increase, see figure 4.12 a). The shunt resistor is very low, 25 mΩ, so a small increase in resistance will give a percentage wise large error. On the second version of the back panel, correct kelvin connection was implemented, see figure 4.12 b).

4.5.2 Current and Voltage Sensor

The typical error of the INA226 voltage measurement was computed to be 0.044% while the Fluke 289 has a typical voltage measurement error of 0.025%. When including the uncertainty due to the typical error we measured 3388.2±0.9 mV with the Fluke 289 and measured 3388±2 mV with the INA226.

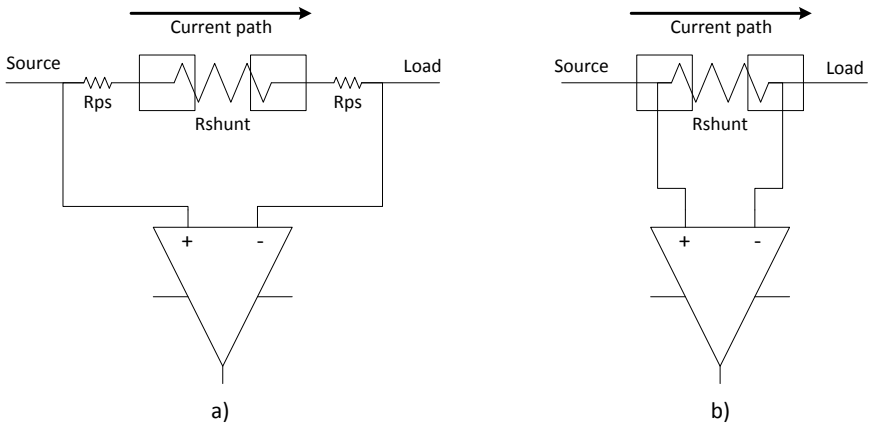


Figure 4.12: a) Non-Kelvin connection: Parasitic series resistance (Rps) is causing current measurement error. b) Correct Kelvin connection: PCB traces to opamp need to start exactly on the shunt resistor pads.

The typical error of the INA226 current measurement is 1.0% or ±1 mA, whichever is greater. Different currents was measured and compared with the Fluke 289 which has a typical current error of 0.15%. When including the uncertainty due to the typical error we measured 88.4±0.1 mA with the Fluke 289 and 89±1 mA with the INA226.

A slight deviation from the optimal kelvin connection on the shunt resistors was discovered, which results in an extra parasitic series resistance of approx. 0.5 mΩ. This will only be noticeable when measuring large currents. We got 601.8 mA from the Fluke 289 and 610 mA from the INA226. 610 mA is above the typical error of 1.0% and the reason is the deviation from kelvin connection. This is not critical because the error is still small, approx. 1.3% instead of 1.0%.

4.5.3 Temperature Sensor

The typical accuracy of the TMP175 is ±0.5°C, but due to the shift to 8 bit the typical accuracy becomes ±1°C. The Fluke 289 with a thermocouple sensor, 80PK-1, was used to check if the TMP175 works as expected. The typical accuracy of the 80PK-1 is poorer (±1.1°C) but we will get a feeling whether the TMP175 operates within the expected range or not. First the Fluke measured 0.6±1.1°C and the TMP175 measured 0±1°C. Then the Fluke measured 23.4±1.1°C while the TMP175 measured 23±1°C. Finally the Fluke measured 50.5±1.1°C and the TMP175 measured 50±1°C.

4.5.4 Coulomb Counter

The worst-case error of the coulomb counter was computed to be 3.64%. Two different tests were performed to characterize the coulomb counter. First a short time test, which was drawing a fixed current for 5 minutes and compare the result from STC3100 with the computed result. A 10 Ω resistor was connected with the Fluke 289 in series. The current

drawn by the load was measured to be 271 mA. We have to add the current consumption by the EPS, which was measured by the Fluke 289 to be 22 mA in *SENSOR* mode. In total this gives 293 mA for 5 minutes which results in

$$\begin{aligned} SoC &= 293\,\text{mA} \cdot \frac{5\,\text{min}}{60\,\text{min/h}} \\ &= 24.4\,\text{mAh}. \end{aligned}$$

The charge measured by the STC3100 was 24 mAh which is within the expected accuracy range.

Earlier we discussed all the error sources of coulomb counting. Especially the offset which builds up over time is a concern. To get a feeling of the offset the system ran for three days. The battery was charged to 100% and the coulomb counter set to zero. Each day the battery was recharged up to 100% without resetting the coulomb counter. The result showed a negative offset building up. This offset is to a small degree dependent on the current drawn from the battery. The battery SoC error was around -8 to -12 mAh each day. In percentage of the total battery capacity, this is not a big error. In one month the error will accumulate to around 5%. As long as the ground station crew is aware of this offset error, it should not be a big problem. As discussed earlier, this measurement is not used to take critical actions, it is just for informational purposes. The ground station crew can remove this offset by issuing a reset command of the coulomb counter charge register.

4.6 Firmware Testing

The firmware has been tested by connecting the EPS, OBC and COMM together. All the different slave commands have been tested and found working. This has been tested both with the OBC sending commands and with a Labview setup using a NI USB-8451 with I^2C interface. The commands were sent from a Labview program made by Myrland and the response from the EPS was stored in a text-file. It has been verified that the status and diagnostic flags are correctly set before sent to the OBC. When sending illegal commands the command is not acknowledged, the error is logged and returned to the OBC.

To see how the firmware reacts to a sensor not responding, sensors have been removed from the bus. The sensor ID is logged and sent to the OBC and the MCU initiates a power-toggle of the sensors to try to recover the faulty sensor. The I^2C line has been pulled low with the result that all the sensors become unavailable. It has been verified that the MCU does not halt but log the error. It has been verified that the battery warning flag is set and cleared at the expected voltages and the TCS is enabled and disabled at the correct temperatures.

The firmware has been running for many hours in different temperatures to look for random and unexplainable errors. The system has been stable and has not shown any sign of unexplainable errors.

4.6.1 System Clock

It is important to find the optimum clock rate for the MCU to avoid wasting power. To find the optimum clock rate, the time spent on the housekeeping task was measured for different system clock rates. The time was measured by comparing the RTC register value before and after the housekeeping task. See figure 4.13 for a plot of the results. We see that time is falling exponentially towards a minimum time of approx. 8 ms. Due

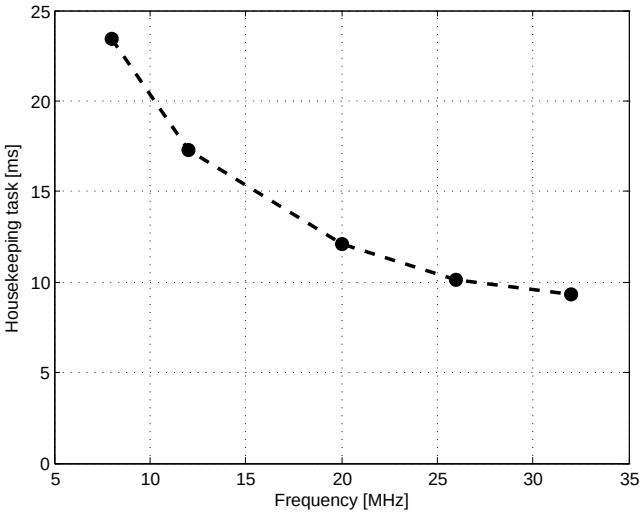


Figure 4.13: Time spent on the housekeeping task vs system clock rate.

to all the I^2C transactions, the housekeeping task will take at least 8 ms no matter how fast clock rate we use. We see that we need 17 ms to finish the housekeeping task at 12 MHz. To make an example, we say that the MCU is using 10 mW power at 12 MHz. The power usage is approximately doubled when the clock rate is doubled. This means 20 mW at 24 MHz. See table 4.1 for the total power spent on the housekeeping task. From the results we can see that there is no help in increasing the clock rate above 12

Table 4.1: Power usage during housekeeping task

Clock rate [MHz]	Power [mW]	Task time [ms]	Total power [mWs]
8	6.67	24	0.16
12	10	17	0.17
24	20	11	0.22

MHz to lower the power usage. The power saved by decreasing the clock rate from 12 MHz to 8 MHz is very little. To maintain the responsiveness of the system in case of critical situations a system clock rate of 12 MHz has been chosen.

Chapter 5

Discussion and Conclusion

5.1 Electrical Power

The power system design described in this thesis is based on the SPV1040 charger. SPV1040 incorporates both CC-CV battery charging and MPPT algorithm which give a compact and simple charging system. Test results show that the charger works as promised by the manufacturer. A charger efficiency of 90% has been achieved which is close to the maximum efficiency of 91-92%. The charger reacts quickly to changes in solar conditions giving maximum power at all times. An internal zero crossing detector prevents reverse current into the charger and the blocking diode (SPV1001N30) prevent reverse current into shaded PV-cells. The only concern is how the charger will react to radiation in space because SPV1040 is not space qualified. Otherwise the charging system should be quite robust and it is running independently of the rest of the power system. If the battery voltage reaches the satellite brownout level of 2.35 V, all the satellite systems shut down except the charging system. Four separate chargers are implemented to minimize the consequences of a charger failure. The charger design fulfils the requirement of continuous power supply to the satellite in orbit.

The A123 $LiFePO_4$ battery cell has been chosen for electrical energy storage. The main benefits of using these batteries are the abuse tolerance, low self-discharge during storage and high depth of discharge count. From tests performed, the battery temperature must be above -10°C to avoid too low battery voltage for normal satellite operation. In case of too low temperatures, an active battery heater of 0.5 W has been implemented to regulate the battery temperature. The thermal inertia of the battery pack has been measured and it is high compared to the back panel. The high thermal inertia will prevent rapid temperature changes on the battery pack. This tells us the battery heater is probably not needed. The battery heater is default off and has to be enabled by the ground station crew in case it is needed. Earlier work on the electrical power system suggested distributing unregulated power. After analyzing the thermal challenges it has been decided to stick with unregulated power distribution. Each subsystem is responsible for regulating the power with respect to their requirements.

5.2 Power and Communication Bus

The power bus structure is implemented using a current-limited power-distribution switch, TPS2557. This switch has been tested and it performs as promised by the manufacturer. Auto-retry functionality has been implemented to be able to remove SELs fast and automatically. This auto-retry functionality has been tested and works as expected.

The communication bus is implemented using the LTC4301 I^2C buffer. This buffer solves the problem with power through pull-up resistors and different I/O levels on the CubeSTAR subsystems. The communication bus has been tested between the EPS, OBC and COMM module and no problems were found.

5.3 Sensors

Sensors allowing the system to monitor current, voltage, temperature and battery SoC have been implemented. The most important sensors are the battery voltage and temperature sensors. These sensors are implemented using TMR because these measurements are used to change the state of the satellite. The TMR voting system is implemented on the EPS MCU.

The measurement error of each sensor has been calculated and tests show that the sensors operate within the expected accuracy range, except a small deviation from kelvin-connection on the current sensors. As anticipated, the coulomb counter accumulates an offset. The offset is small but will grow large over several weeks. This offset can be removed by issuing the reset command of the coulomb counter. The measurement accuracy of the different sensors is considered to be sufficient for our application.

5.4 Microcontroller

An XMEGA128A1U is implemented to control the power system. The hardware support and computational strength of this MCU is found to be sufficient for our application. The MCU is the most critical part in the EPS when it comes to reliability and fault tolerance. Due to the requirement of designing simple systems, none of the CubeSTAR subsystems implement redundant processing systems. Other measures are taken to increase the fault tolerance.

Temporary variables are stored in the SRAM data memory which is vulnerable to SEUs. Hamming codes are implemented to be able to detect erroneous commands from the OBC and to protect the status and diagnostic flags. A watchdog timer and reset command from the OBC is implemented to reset the MCU in case of erroneous behaviour. A reset will cause a re-initialization of the MCU, thus rewriting the SRAM and removing bitflips.

The program code is stored in flash memory. If the flash memory suffers from SEUs, a reset will not help to remove bitflips. As discussed earlier the flash memory of the XMEGA is more radiation tolerant compared to the SRAM, so the probability of SEUs

in flash is considered to be low. At an early stage of the design process it was discussed whether the OBC should be able to re-program the EPS and other subsystems while deployed in orbit. This has not been implemented because of the extra complexity introduced.

In case of an MCU failure, OBC and COMM are default enabled using external pull-ups. This works if the EPS XMEGA pins are tri-stated (high impedance) as they do when powered down, but this is not guaranteed to happen with every kind of processor failure. If the XMEGA choose to set random logic values on the output pins, we lose control of the subsystem powering scheme.

As discussed earlier, a SEL can only be removed by a power-toggle. The different modules and sensors are automatically power-toggled but at the time of writing this is not implemented on the EPS MCU. To increase the SEL tolerance a TPS2553 in front of the 3.3 V regulator is recommended. The 3.3 V regulator limits the current to 400 mA during short-circuit and the minimum current limit of the TPS2557 is 500 mA, so TPS2553 is needed instead of TPS2557. TPS2553 is the same switch but with a current limit range of 75-1700 mA.

The firmware has been designed with focus on simplicity because a complex system will introduce an increased risk of errors. In addition it will be easier for other to understand and maintain the source code. The EPS will only interfere with the satellite operation when receiving commands from the master unit or when the battery voltage or temperature is too low. All commands have been tested and found working. The system has been tested for many hours both in temperatures from -20° to +50°C and in vacuum (5 μbar). The system has been stable and has not shown any sign of unexplainable errors.

5.5 Future Work

Some tasks are still remaining, mainly because some of the subsystems are not finished yet. The following list of tasks should be solved before launching the satellite:

- The power consumption of each subsystem must be measured and the current limit level of each module must be adjusted accordingly. The current limit of the OBC module and the EPS sensors is expected to be well below 500 mA. A change from TPS2557 (500 - 5000 mA) to TPS2553 (75 - 1700 mA) on these two systems should be considered to improve the SEL tolerance.

- The time delay needed to power-toggle the subsystems must be computed. We need to know how much capacitance the subsystems add to their power bus to compute the worst-case power-toggle time.

- To increase the SEL tolerance on the EPS, it should be discussed again whether to implement automatic power-toggling of the EPS. A TPS2553 switch with auto-retry before the 3.3 V regulator is recommended.

- The current sensor shunt resistor connection can be optimized to reach an even lower measurement error.

- There is no reverse voltage protection on the integration connector and this could possibly damage the satellite beyond repair. The satellite will be handled by the launch provider and possibly other external personnel. Reverse voltage protection diodes in combination with a polarized connector is recommended to make the integration connector foolproof.
- New back panels should mount a 100 Ω instead of 1 kΩ fault pin current limit resistor.
- It should be discussed to increase the battery heater power to e.g. 1 W.
- The final mechanical assembly the battery pack using adhesive to fasten the batteries is not tested.
- A radiation analysis of the various components, like SPV1040 and TPS2557, should be performed.
- A more thorough thermal analysis of the satellite should be performed.
- There should be performed extensive system testing with all subsystems running. These tests could reveal that some of the various thresholds and constants of the EPS should be adjusted. Stress testing should be performed to look for any unknown errors in the EPS source code.

5.6 Summary

This thesis has described the design and implementation of a new CubeSTAR electrical power system. A power system that fulfils the functional requirements has been realized and tested. The system has been realized using COTS components with a minimum operating temperature range from -40° to +85°C. Both the hardware and firmware has been designed with focus on simplicity due to limited space available and to reduce the risk of errors. The final conclusion on whether the power system design is good enough cannot be drawn before we launch the satellite. The system has been tested thoroughly in conditions as similar as possible to the conditions in space and the system has worked as expected so far.

References

[1] A123. High power lithium ion APR18650M1. http://www.gylling.no/produkter/batterier/a123/APR18650M1%20Datasheet%20April%202008.pdf, 2008. Accessed: 14.09.2012.

[2] Ashok Ambardar. *Digital Signal Processing: A Modern Introduction*. Thomson, 2007.

[3] Tore André Bekkeng. Prototype development of a multi-needle langmuir probe system. Master's thesis, University of Oslo, 2009.

[4] Brenton Burnett. The basic physics and design of III-V multijunction solar cells. http://science.gov.tm/projects/soltme/images/database/35_nrel.pdf, 2002. Accessed: 05.12.2012.

[5] California Polytechnic State University. Cubesat design specification rev. 12. http://www.cubesat.org/index.php/documents/developers, 2009. Accessed: 07.09.2012.

[6] Castaner and Silvestre. *Modelling Photovoltaic Systems Using PSpice*. Wiley, 1st edition, 2002.

[7] Clyde Space. Power budgets for mission success. http://www.clyde-space.com/documents/2276, 2011. Accessed: 18.09.2012.

[8] Sam Davis. Electronic design #14302, synchronous rectifiers. http://www.electronicdesign.com/files/29/14302/14302_01.pdf, 2007. Accessed: 17.09.2012.

[9] Thomas Floyd. *Electronic Devices*. Pearson, 8th edition, 2007.

[10] Friedel and McKibbon. Thermal analysis of the cubesat CP3 satellite. http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1054&context=aerosp, 2011. Accessed: 28.09.2012.

[11] Judith Jeevarajan. NASA battery workshop, performance and safety evaluation of high-rate 18650 lithium ironphosphate cells. http://www.2a3a.ru/wp-content/uploads/2011/10/Perf_Safe_18650.pdf, 2009. Accessed: 17.09.2012.

[12] H. A. Kiehne. *Battery Technology Handbook*. Dekker, 2nd edition, 2007.

[13] Søren Bækhøj Kjær. Cubesat power systems. http://www.control.auc.dk/~raf/Aerospace/CUBESAT.pdf, 2002. Accessed: 18.09.2012.

[14] Joakim Myrland. On-board controller & data handling. Master's thesis, University of Oslo, 2013.

[15] Martin Oredsson. Electrical power system for the cubestar nanosatellite. Master's thesis, University of Oslo, 2010.

[16] Henry W. Ott. *Noise Reduction Techniques in Electronic Systems*. Wiley, 2th edition, 1988.

[17] Panasonic. Charging methods. http://www.panasonic.com/industrial/includes/pdf/Panasonic_VRLA_ChargingMethods.pdf, 2005. Accessed: 17.09.2012.

[18] Semig and Wells. A current sensing tutorial - part iii. http://www.eetimes.com/design/industrial-control/4237268, 2005. Accessed: 07.01.2013.

[19] Semig and Wells. A current sensing tutorial - part iv. http://www.http://eetimes.com/design/industrial-control/4237683, 2005. Accessed: 07.01.2013.

[20] Clyde Space. Space qualified batteries. http://www.clyde-space.com/cubesat_shop/batteries/15_integrated-battery-daughter-board, 2011. Accessed: 07.01.2013.

[21] Spectrolab. 28.3% UTJ solar cell. http://www.spectrolab.com/DataSheets/TNJCell/utj3.pdf, 2010. Accessed: 14.09.2012.

[22] ST Microelectronics. AN3064: Using the STC3100 battery monitor for gas gauge applications. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/APPLICATION_NOTE/CD00250837.pdf, 2009. Accessed: 24.09.2012.

[23] ST Microelectronics. AN3319 SPV1040 application note. http://www.st.com/internet/analog/product/251161.jsp, 2011. Accessed: 07.09.2012.

[24] ST Microelectronics. SPV1040: High efficiency solar battery charger with embedded MPPT. http://www.st.com/internet/analog/product/251161.jsp, 2011. Accessed: 07.09.2012.

[25] A123 Systems. Nanophosphate basics. http://www.batteryspace.com/prod-specs/6610_1.pdf. Accessed: 07.12.2012.

[26] A123 Systems. Development of battery packs for space applications. https://batteryworkshop.msfc.nasa.gov/presentations/11_Dev_Batt_Packs_Space_Appl_DCarmen.pdf, 2007. Accessed: 07.12.2012.

[27] Texas Instruments. TPS2557: Precision adjustable current-limited power-distribution switch. http://www.ti.com/product/tps2557, 2008. Accessed: 13.09.2012.

[28] Young and Freedman. *University Physics with Modern Physics.* Pearson, 12th edition, 2008.

Appendix A

Back Panel Schematics

In this section you will find the circuit schematics of the CubeSTAR back panel.

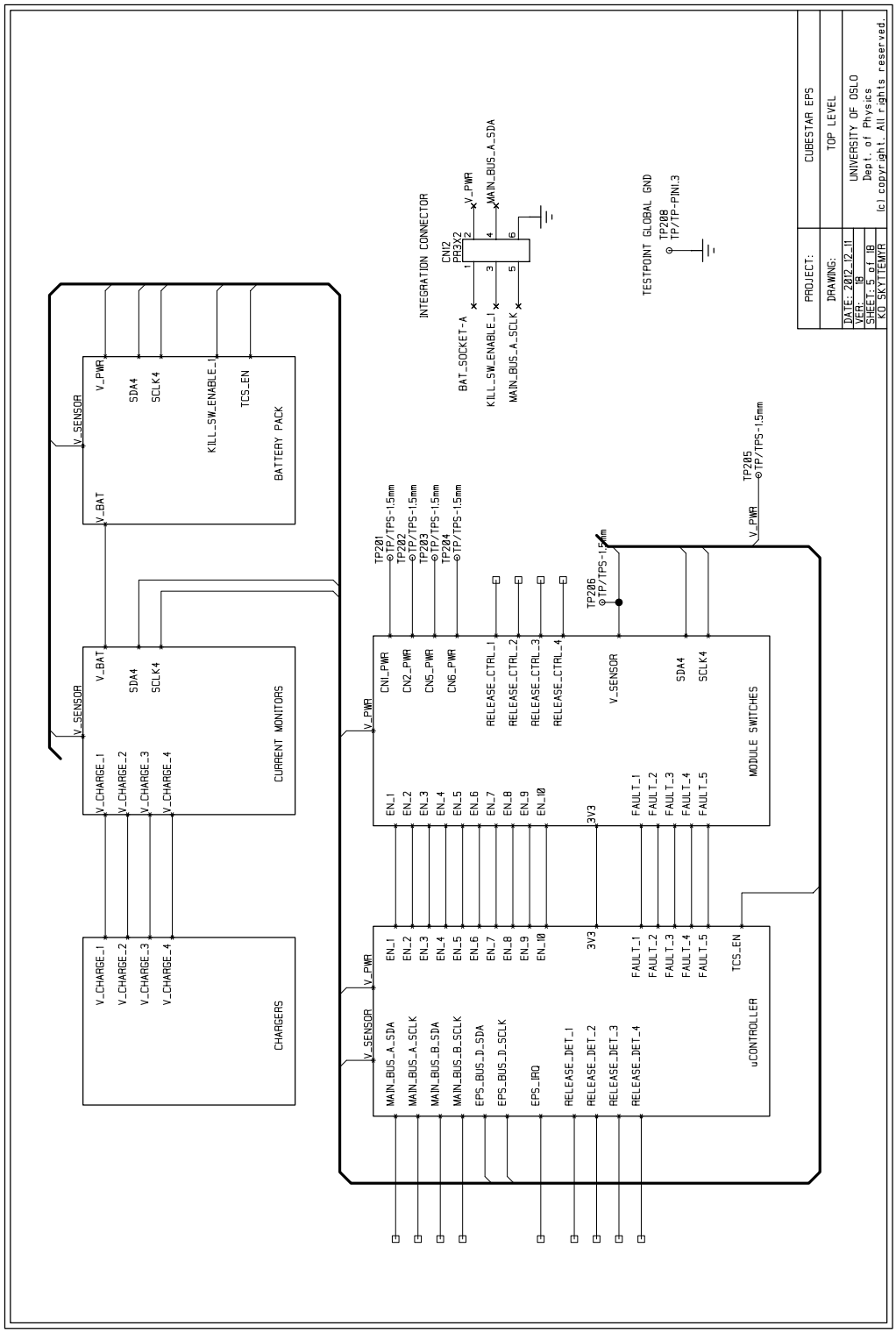


Figure A.1: CubeSTAR backpanel top level

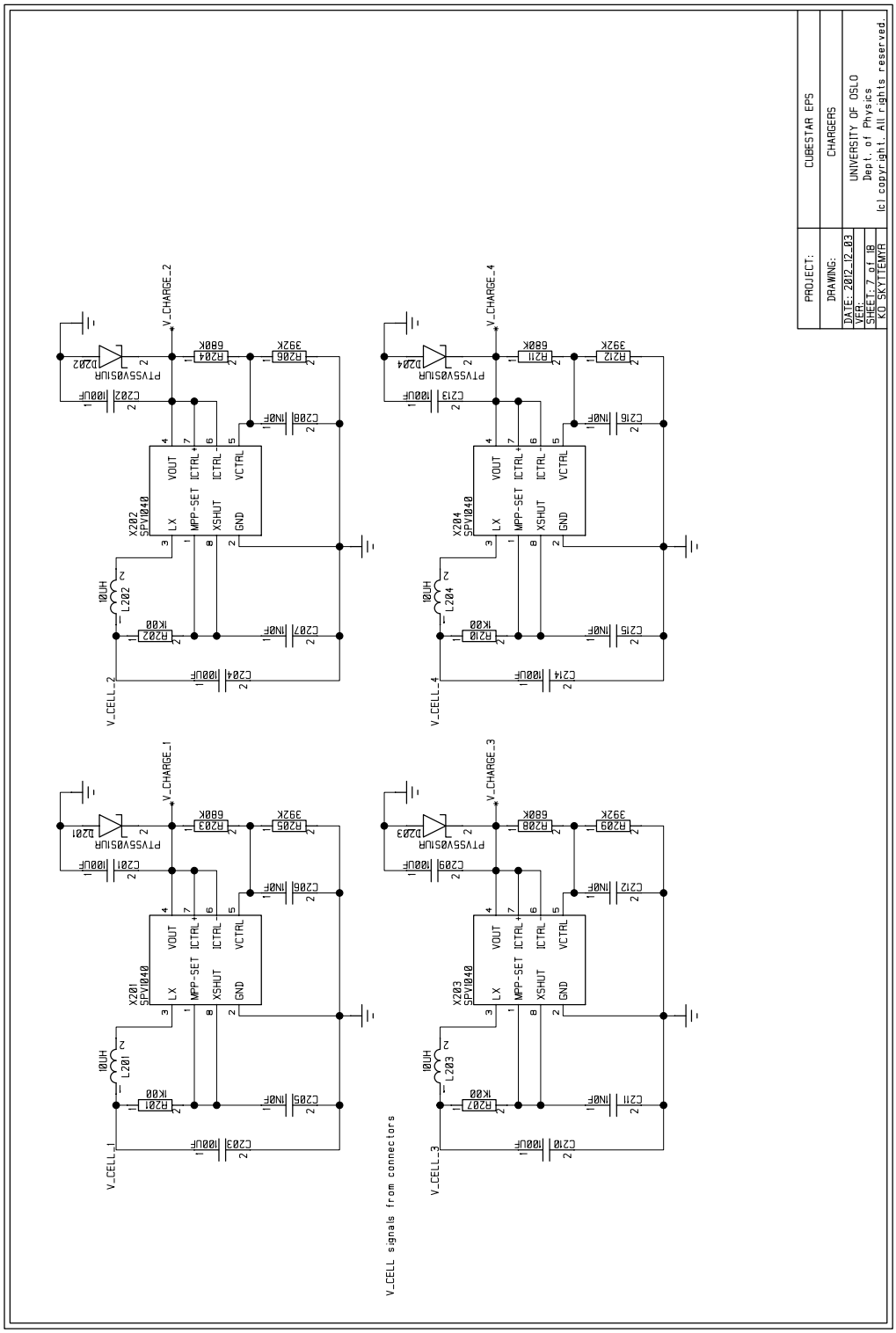


Figure A.2: CubeSTAR backpanel chargers

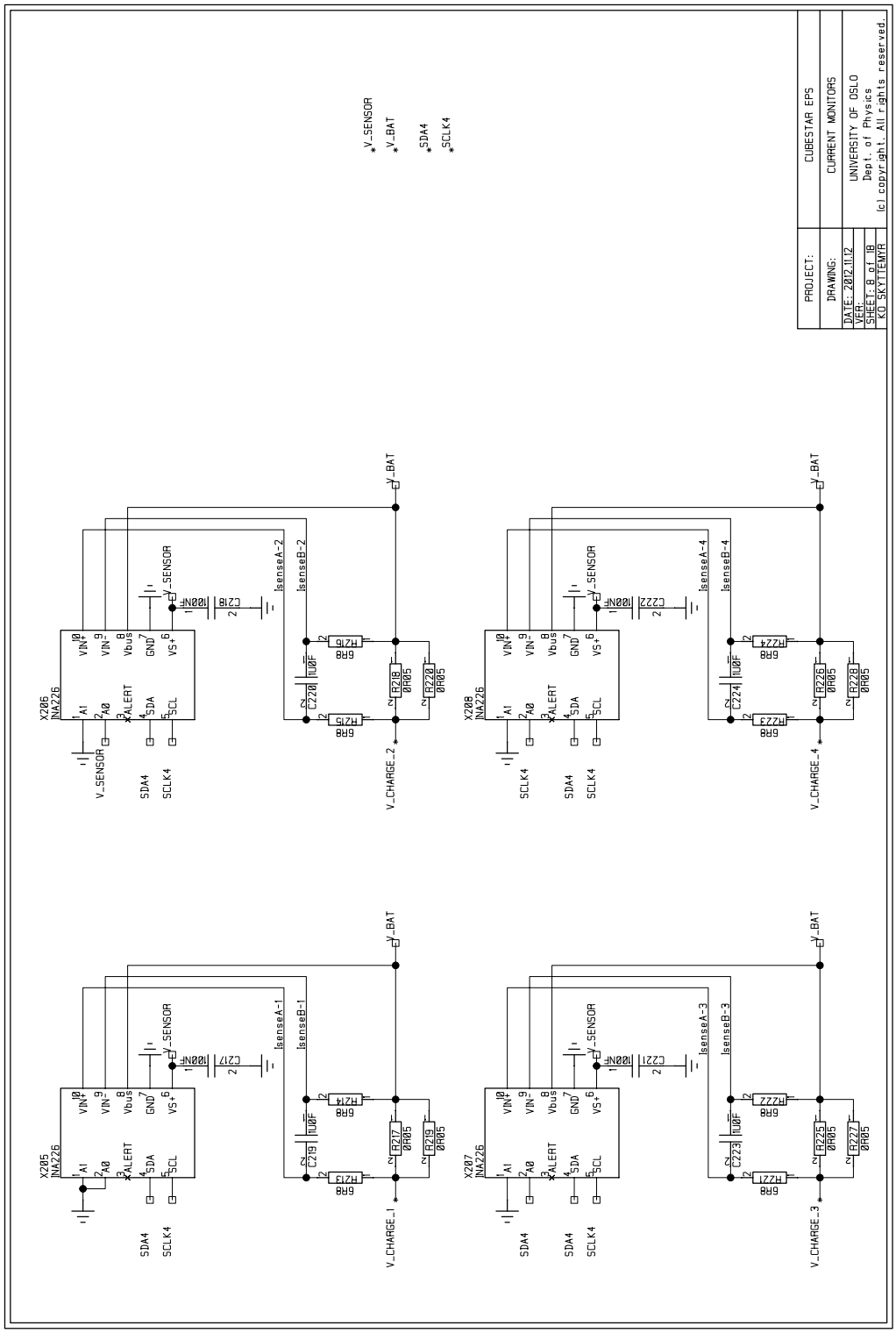


Figure A.3: CubeSTAR backpanel current monitors

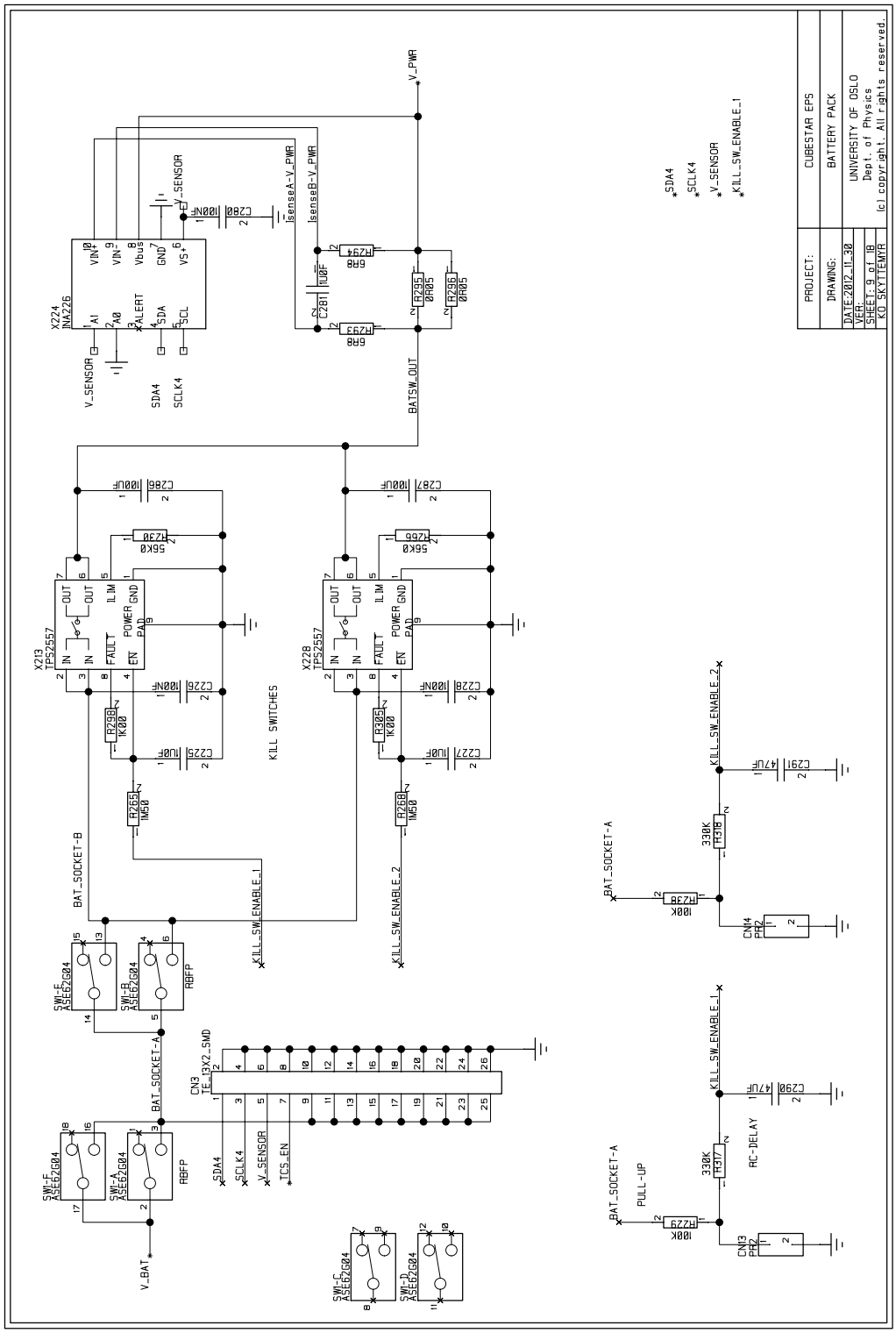


Figure A.4: CubeSTAR backpanel battery pack

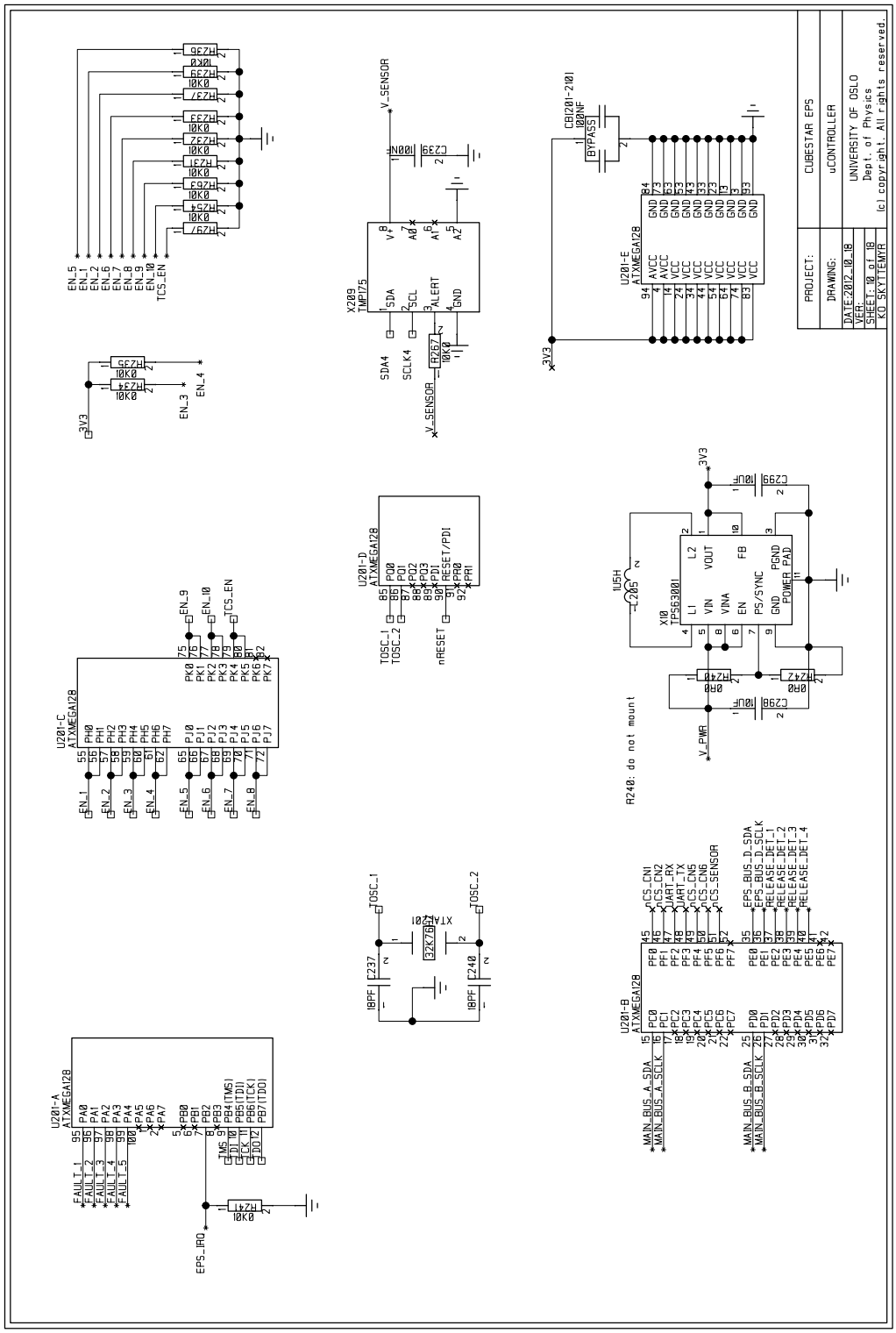


Figure A.5: CubeSTAR backpanel MCU

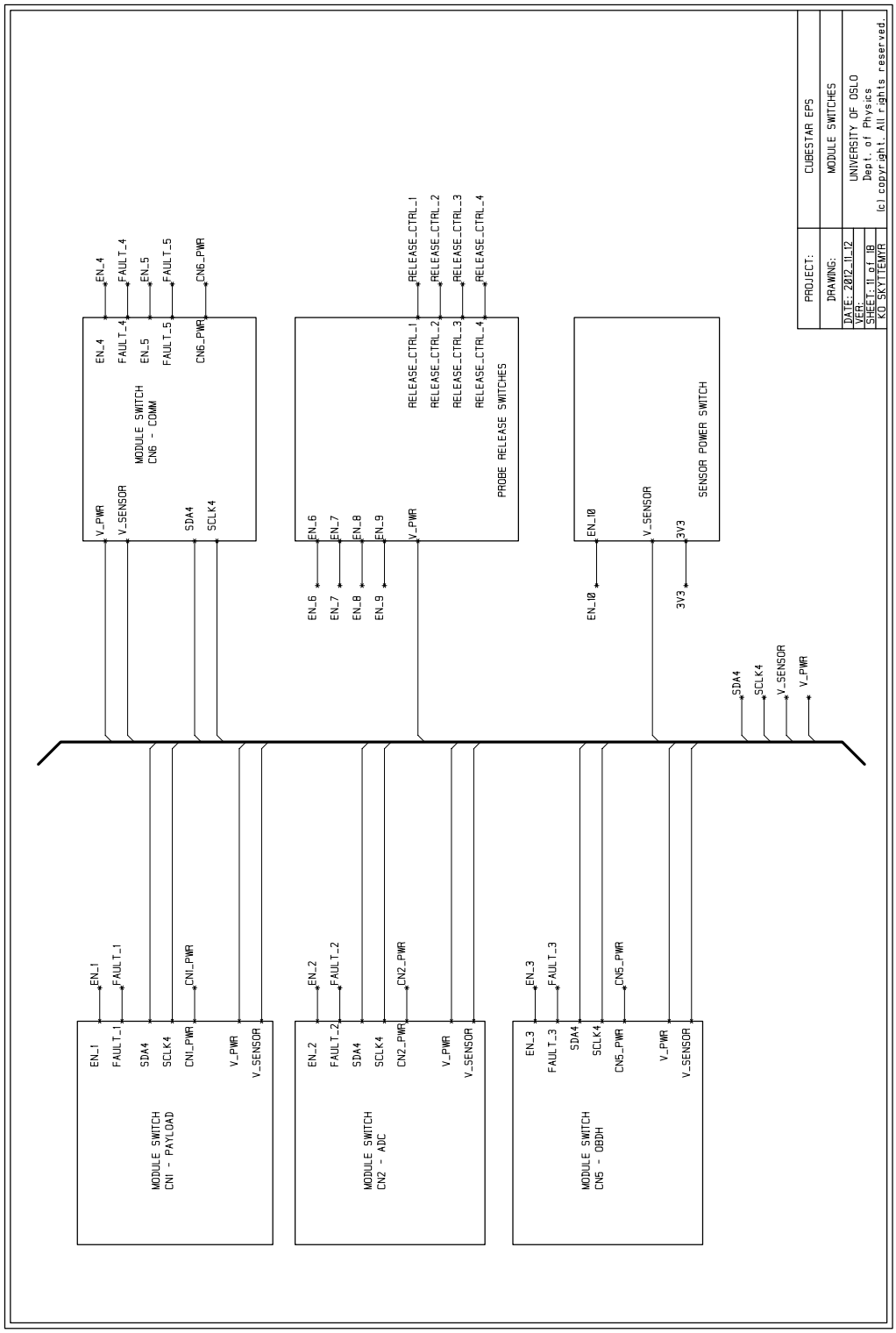


Figure A.6: CubeSTAR backpanel module switches

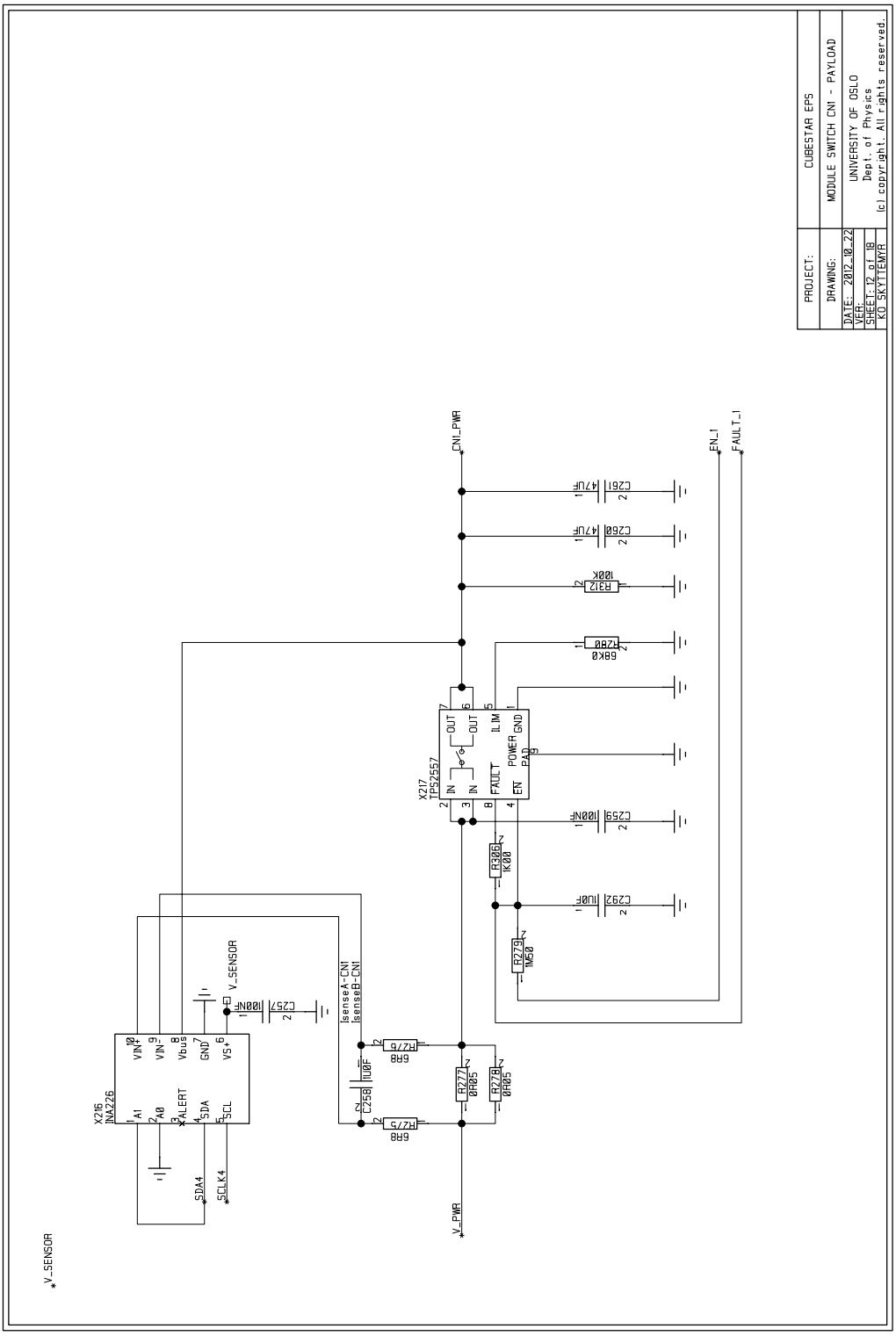


Figure A.7: CubeSTAR backpanel module switch CN1 - Payload

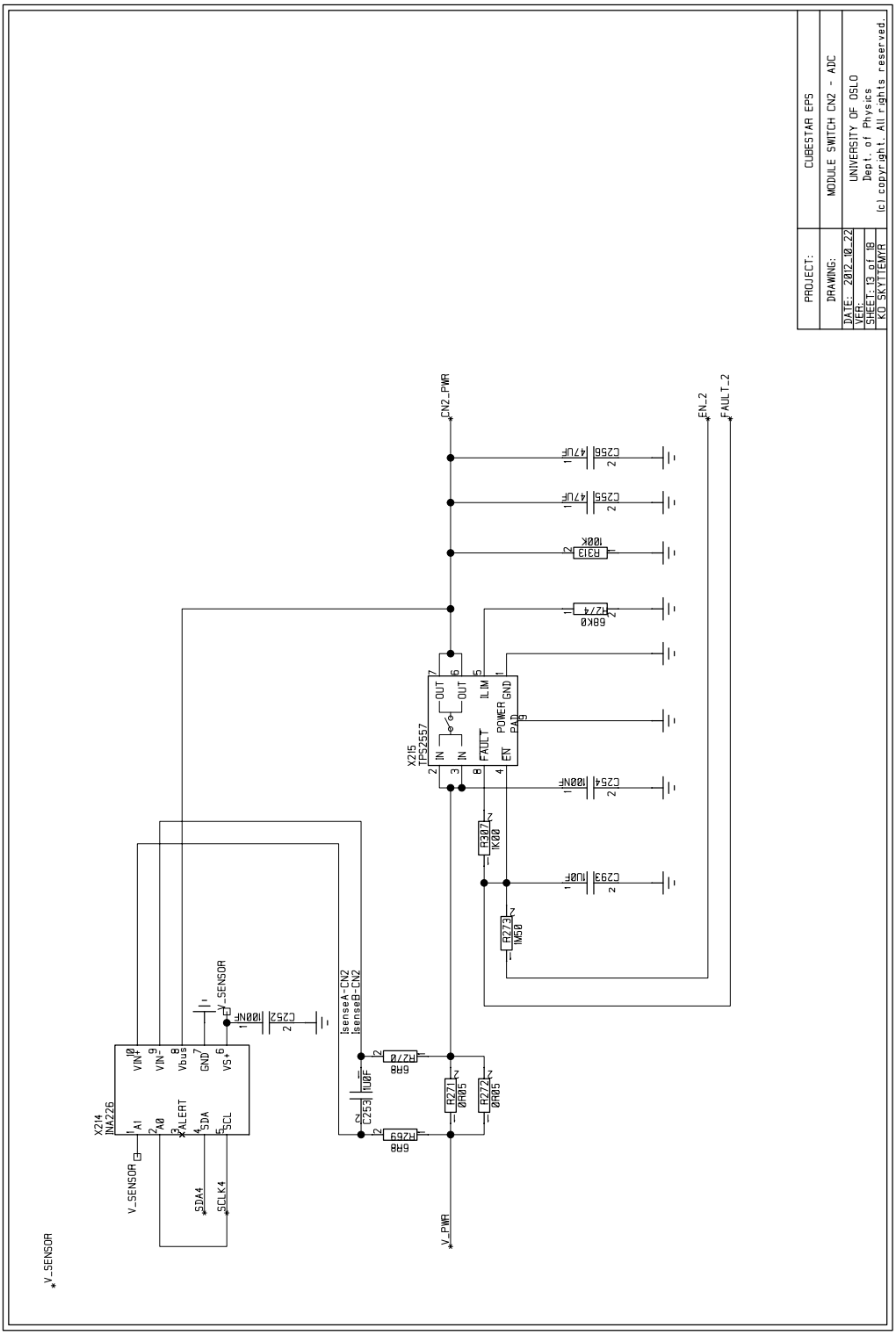


Figure A.8: CubeSTAR backpanel module switch CN2 - ADCS

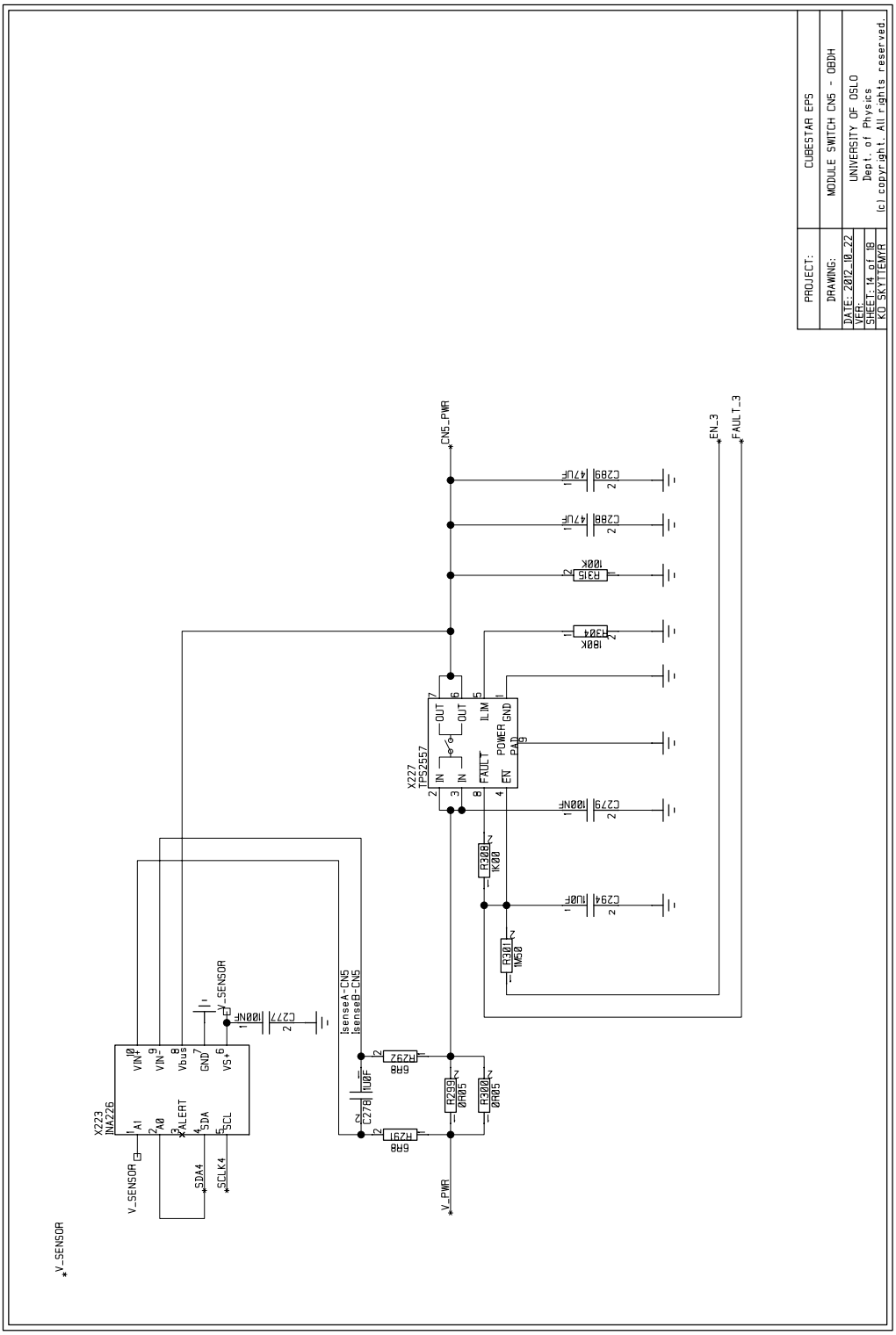


Figure A.9: CubeSTAR backpanel module switch CN5 - OBDH

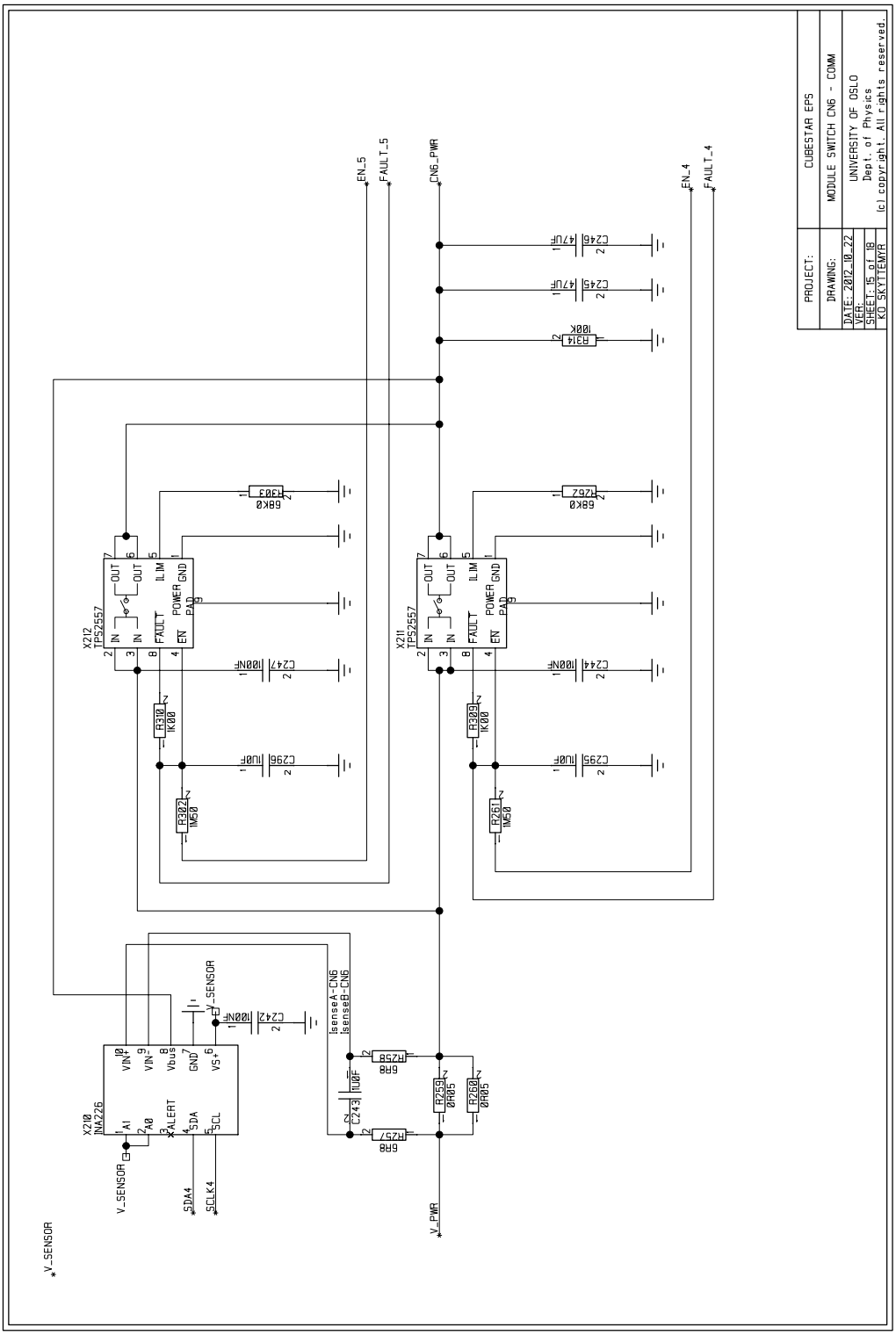


Figure A.10: CubeSTAR backpanel module switch CN6 - COMM

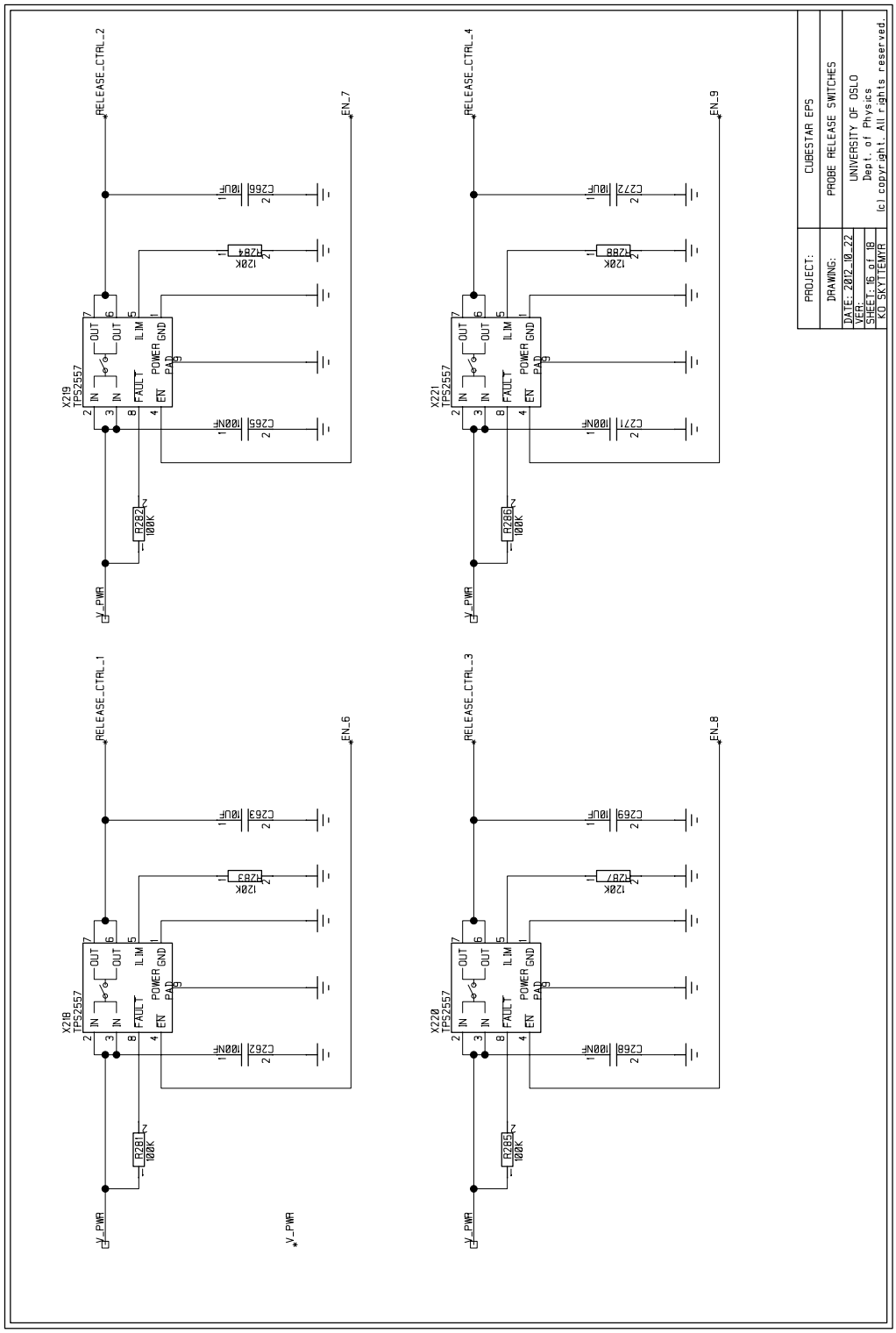


Figure A.11: CubeSTAR backpanel probe release switches

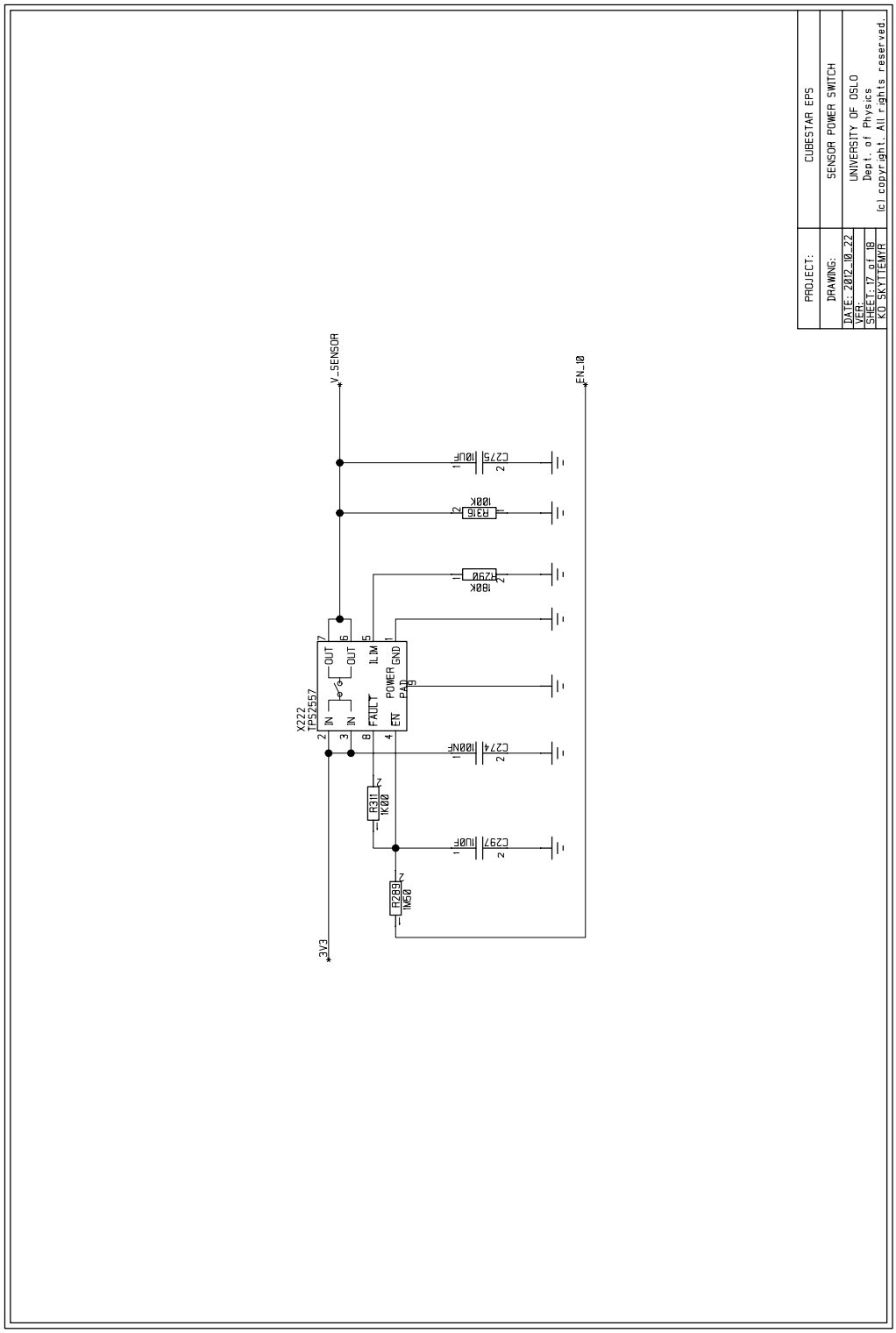


Figure A.12: CubeSTAR backpanel sensor power switch

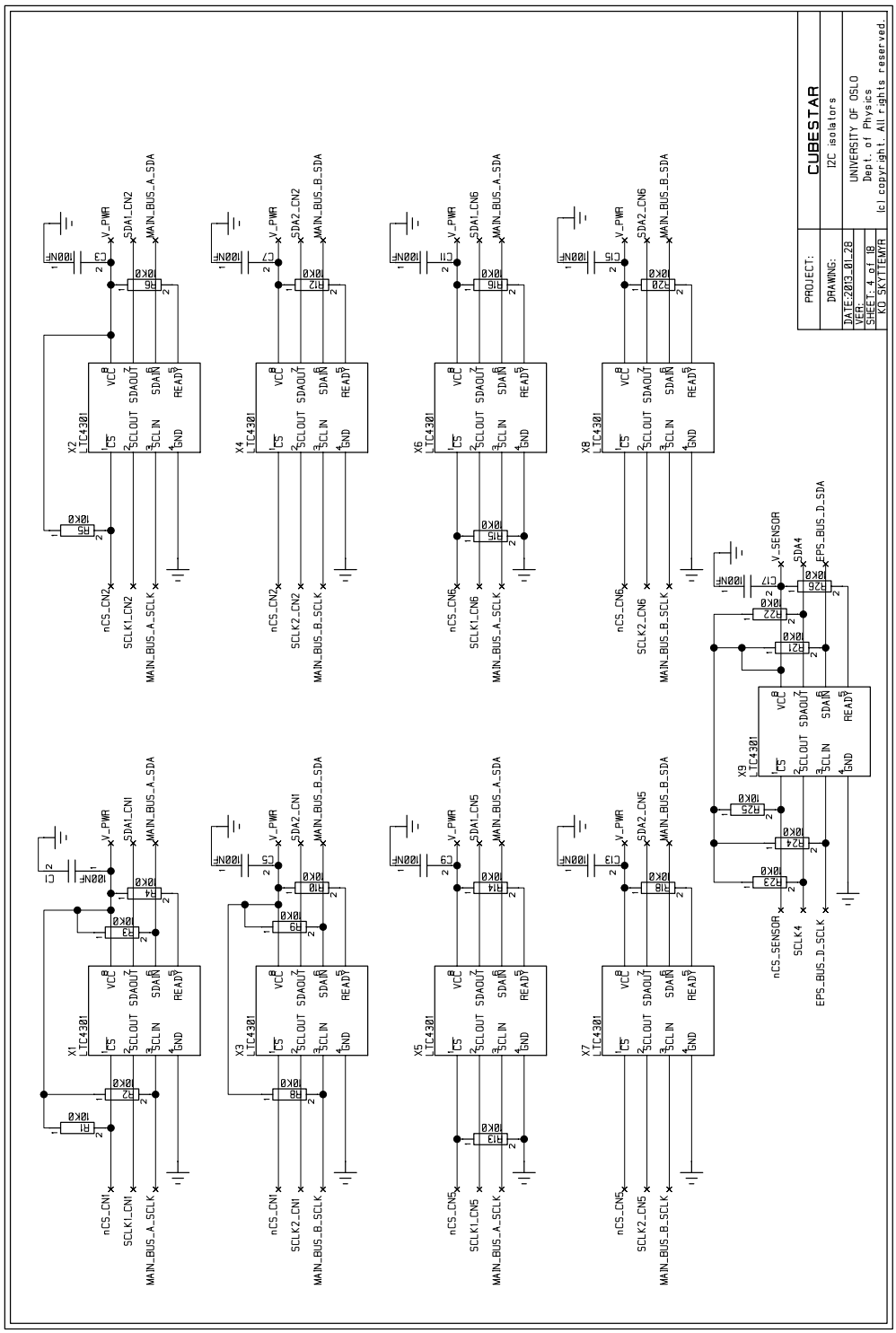


Figure A.13: CubeSTAR backpanel I^2C buffers

Appendix B

Battery Pack PCB

In this section you will find both the circuit schematics and the PCB layout of the battery pack.

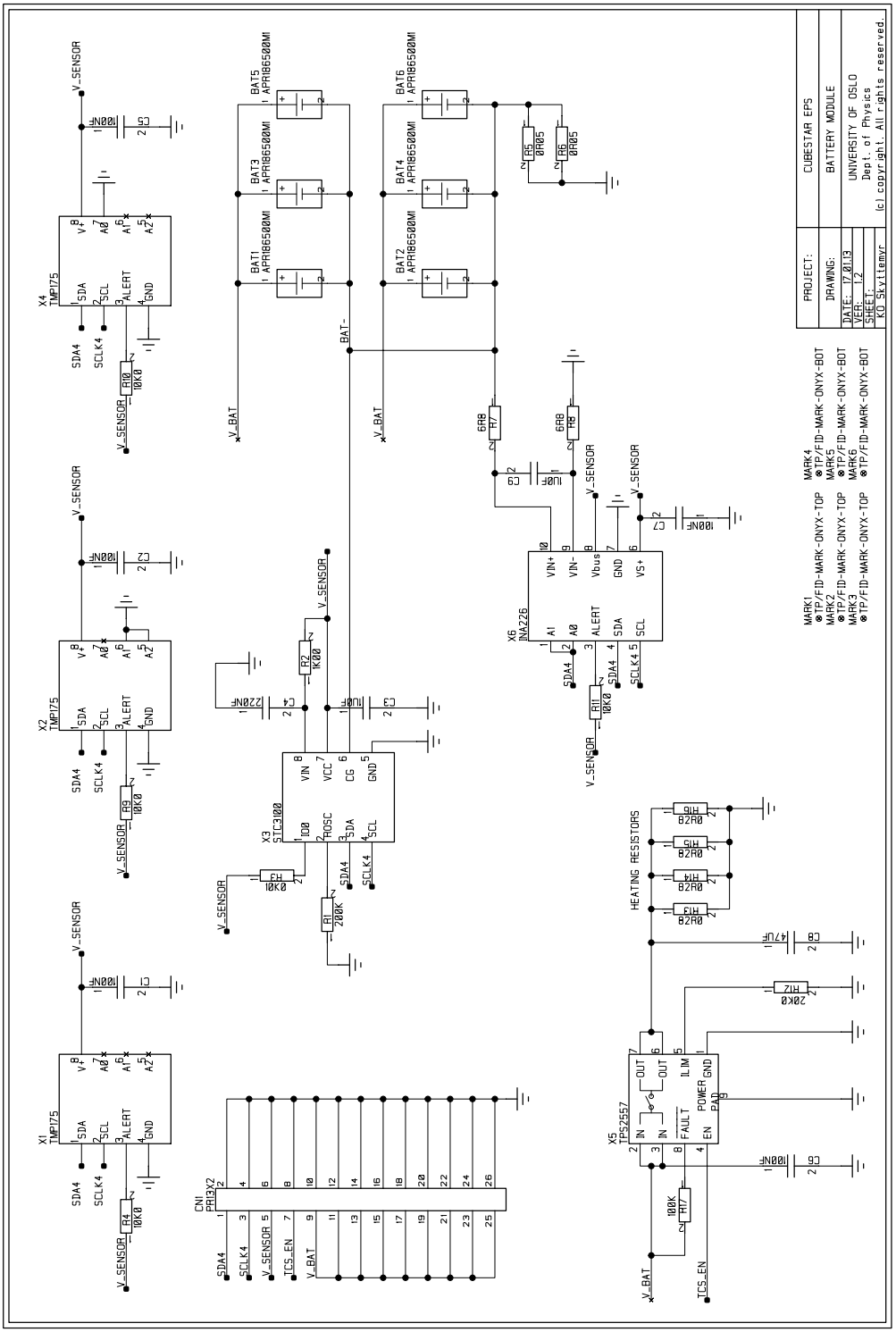


Figure B.1: Battery module schematic

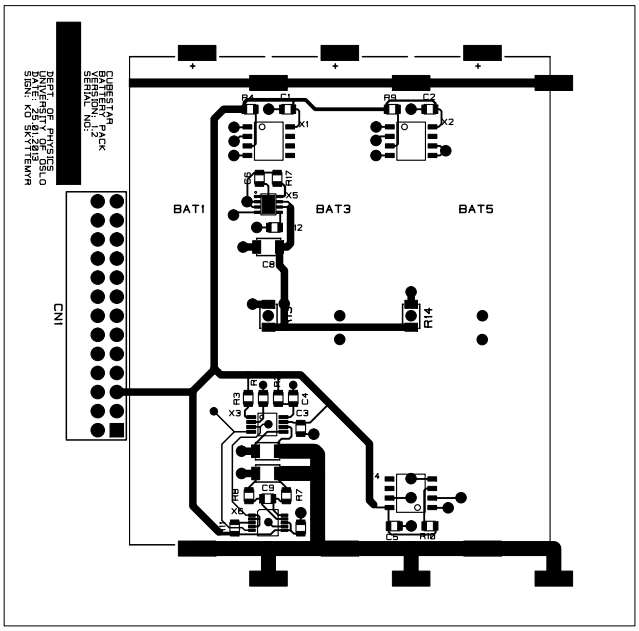


Figure B.2: Battery module PCB top layer

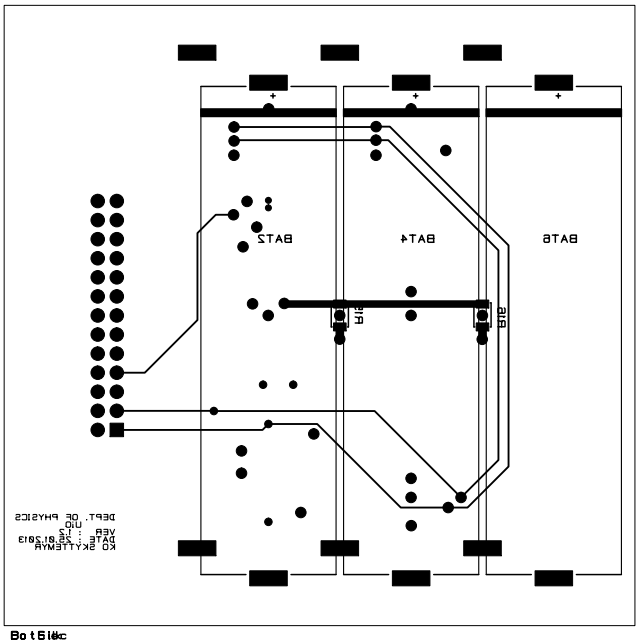


Figure B.3: Battery module PCB bottom layer

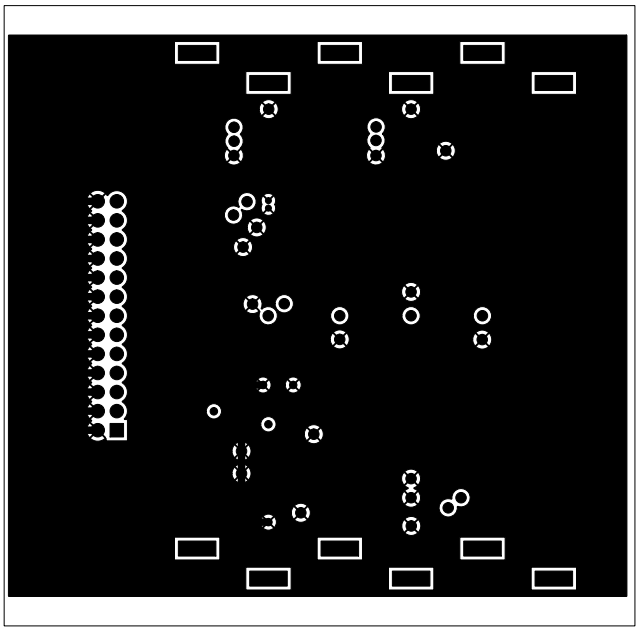


Figure B.4: Battery module PCB ground layer

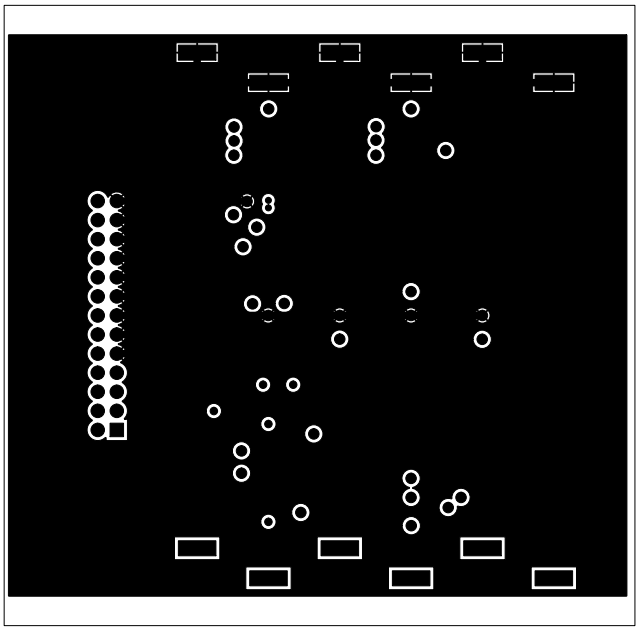


Figure B.5: Battery module PCB power layer

Appendix C

SPV1040 Test Board and Labview DAQ Program

Here you will find the circuit schematics and PCB layout of the SPV1040 test board and the Labview program used to log input and output power from the charger.

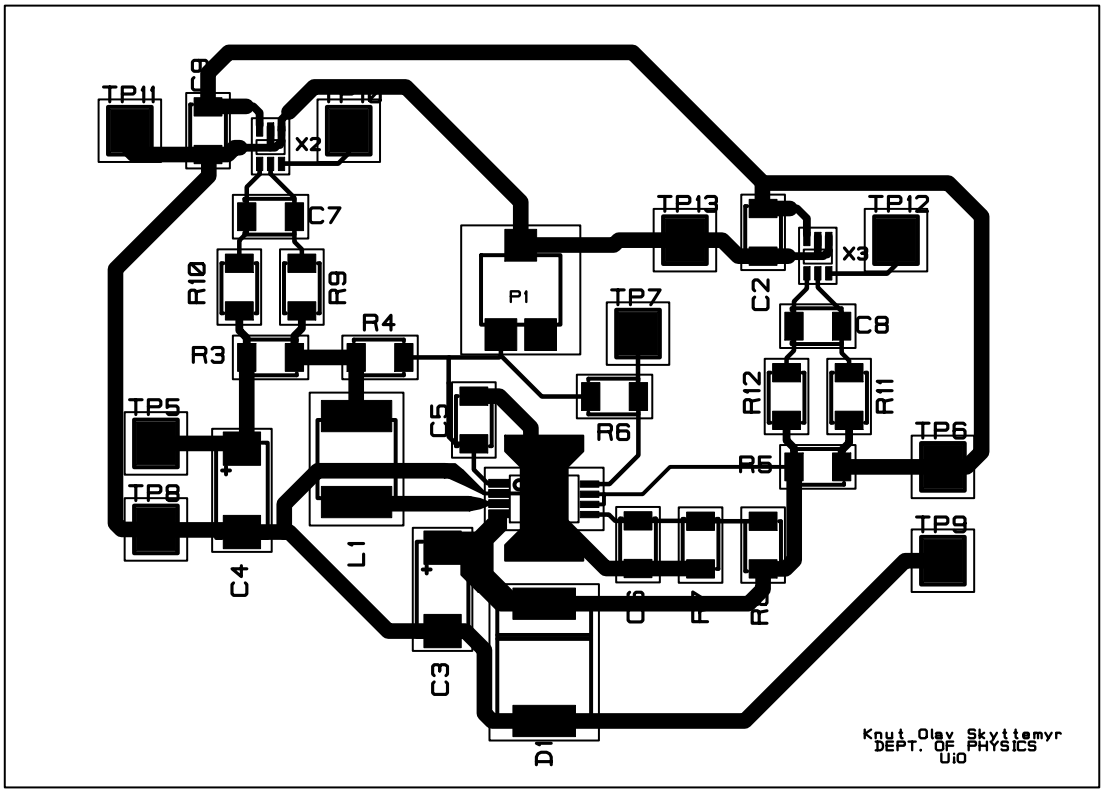


Figure C.1: SPV1040 test board PCB layout.

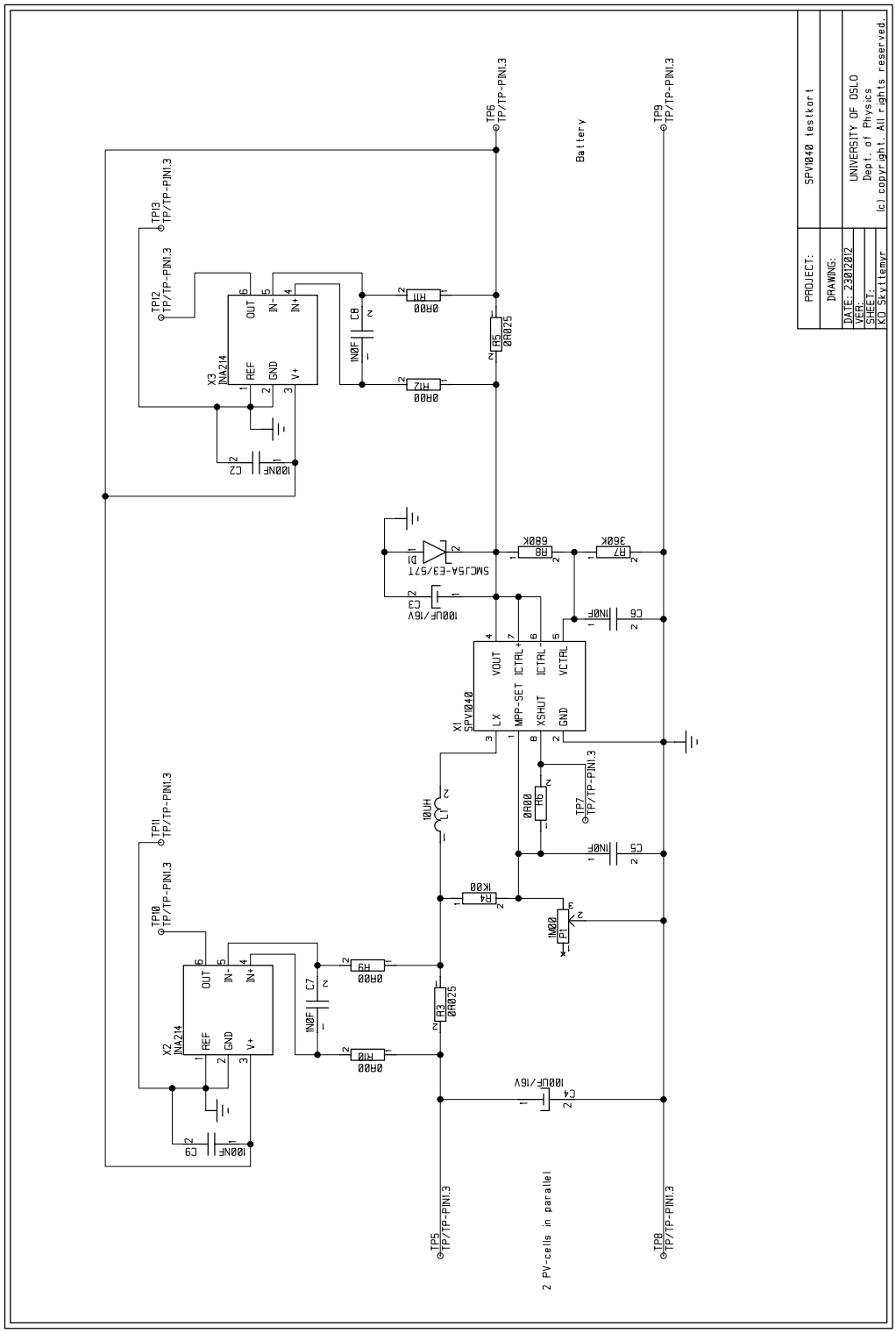


Figure C.2: SPV1040 test board schematic.

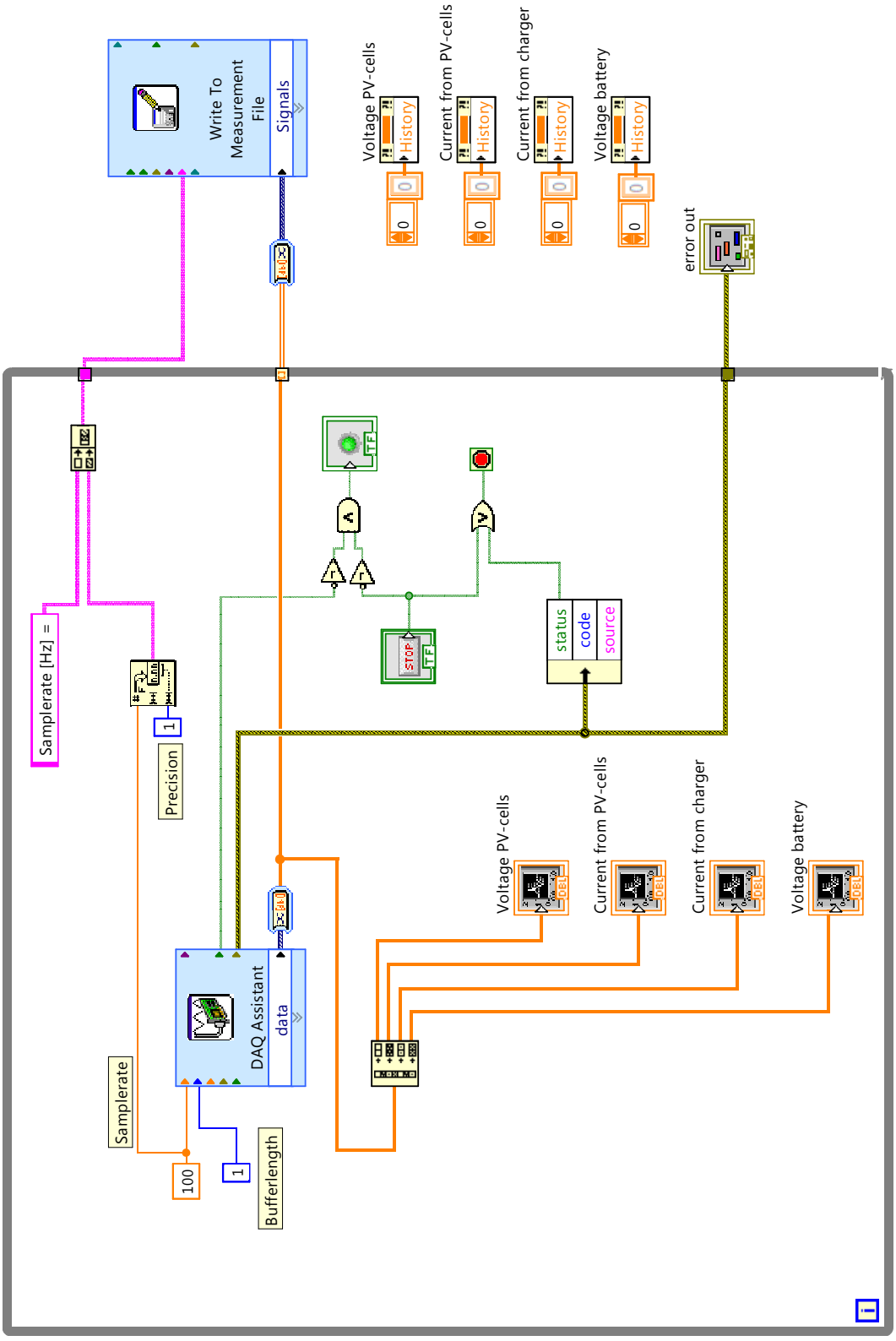


Figure C.3: Labview DAQ program.

Appendix D

Test Result Plots

In this section you will find various test results not shown in the main text.

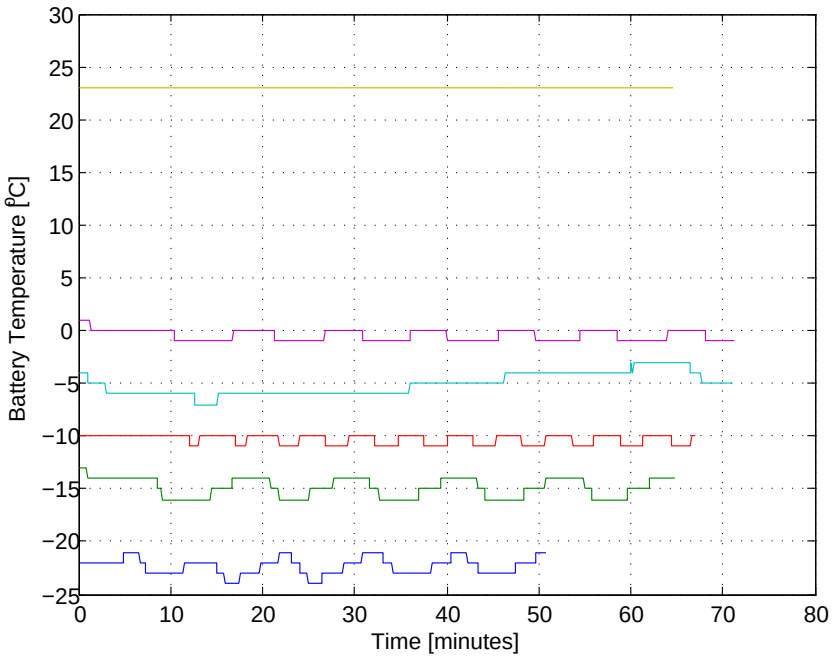
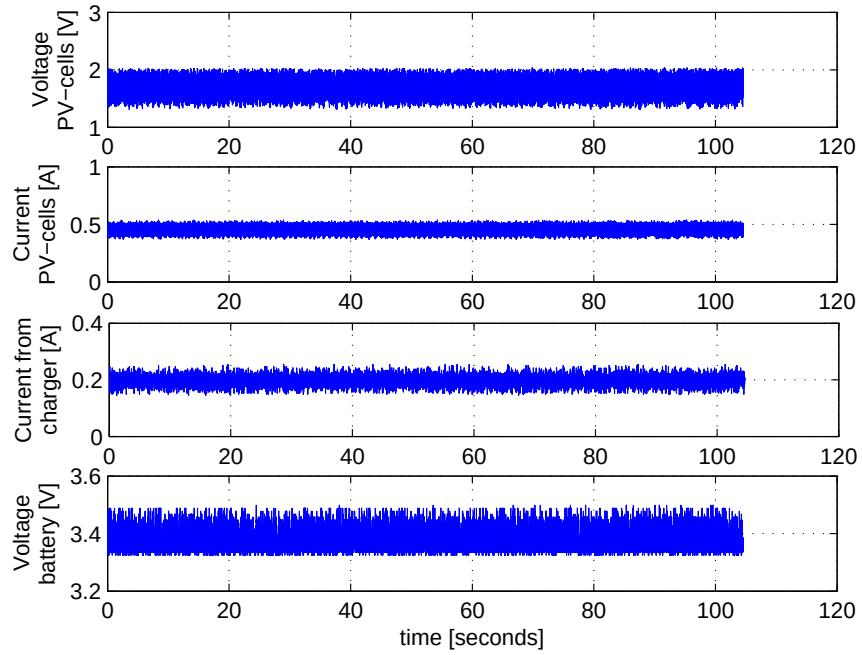
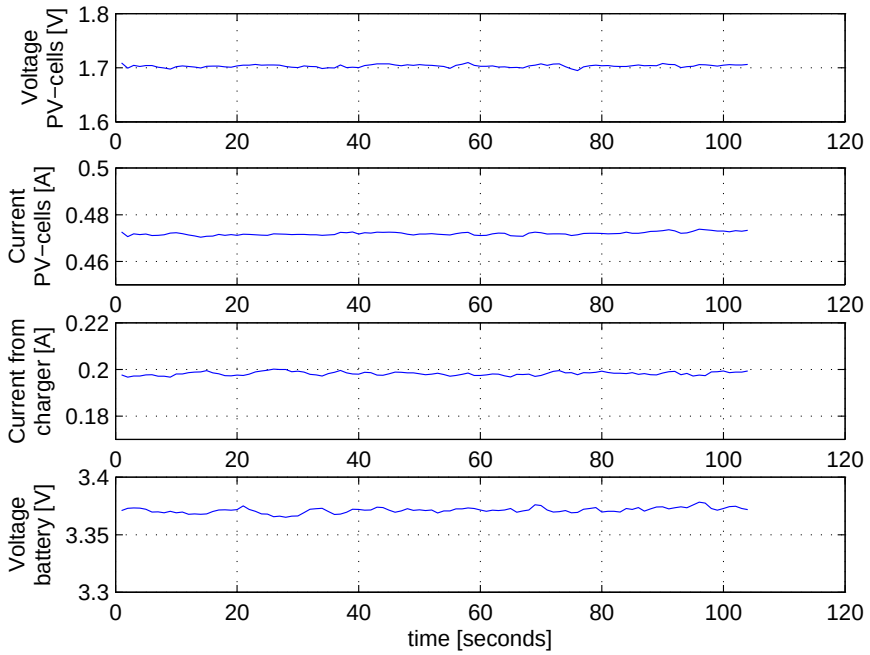


Figure D.1: Temperature data from TMP175 during the battery discharge test. We see how the test chamber struggled with keeping a constant temperature. This has to be kept in mind when reading the battery voltage vs temperature plot.



(D.2a) Filtering off.



(D.2b) Filtering on.

Figure D.2: Showing the effect of arithmetic mean (N=100) and moving averager (N=3) filter. The arithmetic mean filter is good for removing random noise. The moving averager is good for smoothing out the variations caused by the MPPT algorithm.

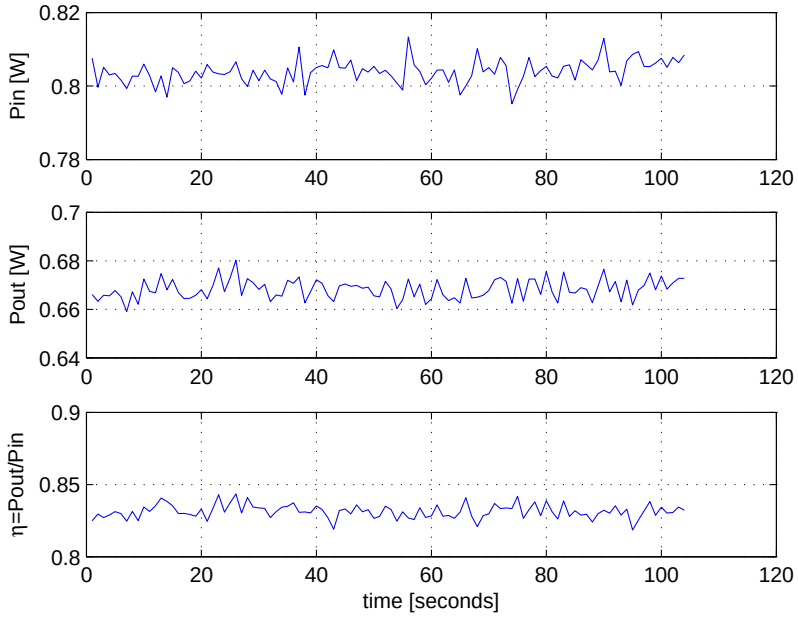


Figure D.3: SPV1040 efficiency measurement, AM and MA filtering on. We can see that the efficiency is 83%.

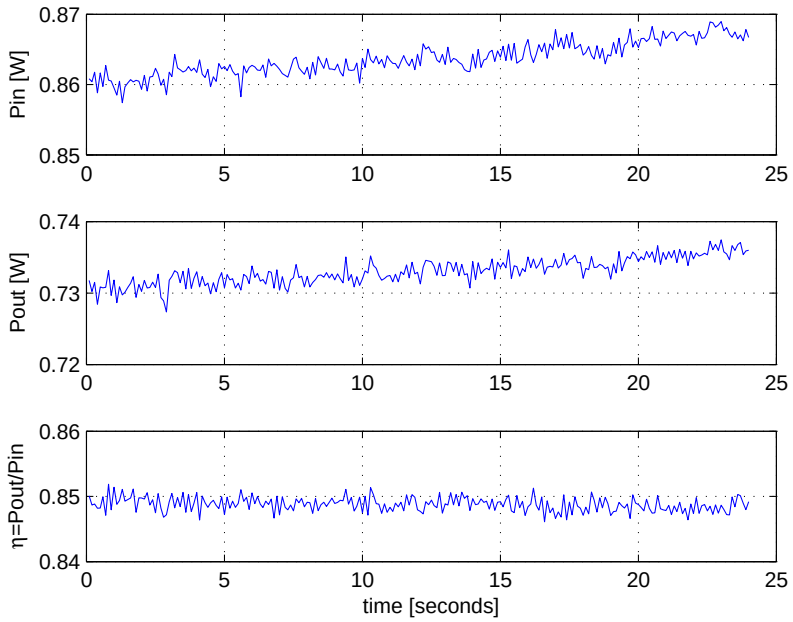


Figure D.4: SPV1040 efficiency measurement with new TVS diode and ceramic capacitors, AM and MA filtering on. The efficiency was increased to 85%.

Appendix E

Microcontroller Source Code

Here you will find the source code for the EPS microcontroller. The source code consists of the following files:

- **EPS.h:** Header with global includes, definitions, variables and prototypes.
- **EPS.c:** main() and ISRs.
- **EPS_init.c:** Initialization of MCU and sensors.
- **EPS_cmd.c:** EPS command handler.
- **housekeeping.c:** Housekeeping functions, fetch and process sensor data.
- **sensor_i2c_cmd.c:** I^2C commands for EPS sensors.
- **uart.c:** UART functions.

Listing E.1: EPS.h

```
1
2  /*
3   *  CubeSTAR EPS
4   *  Filename: EPS.h
5   *  Author: Knut Olav Skyttemyr
6   *  Description: Global includes, definitions, variables and prototypes
7   */
8
9  #ifndef EPS_H_
10 #define EPS_H_
11
12 // System clock
13 #define F_CPU 12000000UL
14
15 // Includes
16 #include <avr/io.h>
17 #include <avr/interrupt.h>
18 #include <avr/eeprom.h>
19 #include <avr/sleep.h>
20 #include <stdlib.h>
21 #include <string.h>
22 #include "ccp.h"
23 #include "sp_driver.h"
24 #include "twi_master_driver.h"
25 #include "twi_slave_driver.h"
26
27 // Global prototypes
28 void TWI_SlaveProcessData(void);
29 void sensor_config(void);
30 uint8_t tmp175_config(uint8_t i2c_address);
31 uint8_t tmp175_fetch(uint8_t i2c_address);
32 uint8_t ina226_config(uint8_t i2c_address, uint8_t mode, uint8_t mask);
33 uint8_t ina226_fetch(uint8_t i2c_address, uint8_t mode);
34 uint8_t stc3100_config(uint8_t i2c_address);
35 uint8_t stc3100_fetch(uint8_t i2c_address);
36 uint8_t stc3100_reset(uint8_t i2c_address);
37 void load_eeprom(void);
38 void error_handling(void);
39 void gather_i2c_status(void);
40 void power_subsystems(uint8_t obdh, uint8_t pld, uint8_t comm, uint8_t adcs,
       uint8_t mode);
41 bool HD4Valid(uint8_t HD4);
42 uint8_t HD4Value(register8_t *value);
43 void clear_statistics(void);
44 void transmit(int8_t data);
45 void transmit_line(int8_t *line);
46 void transmit_hex(int8_t data);
47
48 //
49 //  Definitions and constants
50 //
51
```

```
52 // EPS mode
53 #define SENSOR // SENSOR: Print all sensor data to UART,
54 #define DEBUG  // DEBUG: Print status changes and errors to UART,
55 //#define NORMAL //NORMAL: Normal mode, cannot be active at the same time as
       SENSOR and DEBUG
56
57 // I2C settings
58 #define BAUDRATE    100000 //i2c clock rate
59 #define TWI_BAUDSETTING TWI_BAUD(F_CPU, BAUDRATE)
60 #define I2C_BUS4    TWIE //EPS local I2C bus, master
61 #define I2C_MASTER_vect TWIE_TWIM_vect
62 #define I2C_BUS2    TWID //Secondary I2C bus, slave
63 #define I2C_SLAVE2_vect TWID_TWIS_vect
64 #define I2C_BUS1    TWIC //Main I2C bus, slave
65 #define I2C_SLAVE_vect  TWIC_TWIS_vect
66 #define TIMEOUT_I2C   20 //in ticks of 100us, ~twice as long as the longest
       transaction, default 20
67
68 // EPS slave address
69 #define EPS_I2C_ADDRESS   0x0E
70
71 // Sensor slave addresses
72 #define TMP175_BAT1     0b0110111 //0x37
73 #define TMP175_BAT2     0b0101100 //0x2C
74 #define TMP175_BAT3     0b1110110 //0x76
75 #define TMP175_BACKPANEL  0b0110101 //0x35
76 #define INA226_CHG1     0b1000000 //0x40
77 #define INA226_CHG2     0b1000001 //0x41
78 #define INA226_CHG3     0b1000010 //0x42
79 #define INA226_CHG4     0b1000011 //0x43
80 #define INA226_MAINBUS    0b1000100 //0x44
81 #define INA226_COMM     0b1000101 //0x45
82 #define INA226_OBDH     0b1000110 //0x46
83 #define INA226_ADCS     0b1000111 //0x47
84 #define INA226_PLD      0b1001000 //0x48
85 #define INA226_BAT      0b1001010 //0x4A
86 #define STC3100       0b1110000 //0x70
87
88 #define NR_OF_INA226_1  4 //4, battery monitors
89 #define NR_OF_INA226_2  2 //2, main-bus monitors
90 #define NR_OF_INA226_3  4 //4, module monitors
91 #define NR_OF_TMP175  4 //4
92 #define NR_OF_STC3100 1 //1
93
94 // Interrupt vectors
95 #define FAULT_vect  PORTA_INT0_vect //Fault_1-5 generates interrupt on PORTA
96
97 // UART defines and macros
98 #define UART_PORT   PORTF
99 #define UART      USARTF0
100 #define UART_RXC_vect USARTF0_RXC_vect
101 #define BSCALE_VALUE -7 // 12MHz, 115200 baud. CLK2x = 1
102 #define BSEL_VALUE 1539
```

```
103 #define NEWLINE transmit_line((int8_t*) "\n\r") //UART newline and carriage
        return
104 #define TAB   transmit('\t')          //UART tabulator delimiter
105
106 // EPS constants. Verify all these constants
107 #ifdef SENSOR
108   #define OVF_SEC   5 // 5 seconds default. Number of seconds between each
        RTC overflow interrupt
109 #else
110   #define OVF_SEC   10 //10 seconds default
111 #endif
112 #define MASTER_TIMEOUT  24 // x times 2.5 sec timeout-ticks.
113 #define POWER_TOGGLE_DLY 2000 //delay in ms.
114 #define RELEASE_TIMEOUT 250 //timeout in multiples of 20ms = 5 sec
115 #define V_BAT_WARNING (int16_t) 0x08C0 // 0x0960 * 1.25 = 3000mV, 0x08C0 *
        1.25 = 2800mV
116 #define V_BAT_SAFE    (int16_t) 0x0A50 // 0x0A50 * 1.25 = 3300mV, 0x09B0 *
        1.25 = 3100mV
117 #define V_PWR_MAX   (int16_t) 0x0C80 // 0x0C80 * 1.25 = 4.0 V
118 #define V_PWR_MIN   (int16_t) 0xFFD8 // 0xFFD8 * 1.25 = -50 mV. Must include
        a small negative number due to offset
119 #define VCOMM_MIN   (int16_t) 0x0190 // 0x0190 * 1.25 = 500mV.
120 #define CURRENT_MAX   (int16_t) 0x4E20 // 0x4E20 * 0.1 = 2000mA.
121 #define CURRENT_MIN   (int16_t) 0xB1E0 // 0xB1E0 * 0.1 = -2000mA
122 #define TEMP_MAX    (int8_t) 0x50 //80 degrees
123 #define TEMP_MIN    (int8_t) 0xB0 //-80 degrees
124 #define TCS_ENABLE    (int8_t) 0xF9 //-7 degrees
125 #define TCS_DISABLE   (int8_t) 0xFB //-5 degrees
126 #define TCS_DEFAULT   0 //TCS default off
127
128 enum TPS2557_configuration{
129   ENABLE,
130   DISABLE,
131   TOGGLE
132 };
133
134 enum INA226_configuration{
135   TRIG,   //Triggered conversion mode
136   VC_SENSOR,  //Voltage and current continuous averaging conversion mode
137   C_SENSOR, //Current continuous averaging conversion mode
138   VC_NORMAL,  //Voltage and current continuous conversion mode
139   C_NORMAL, //Current continuous conversion mode
140   PWR_DOWN, //Power down mode, to save power
141   BUL,    //Bus voltage under-voltage alert
142   VOLTAGE,  //Fetch voltage measurement
143   CURRENT,  //Fetch current measurement
144   V_AND_C   //Fetch both voltage and current
145 };
146
147 // EPS slave command list
148 #define EPS_STATUS      0x1E //verify the order of the commands
149 #define EPS_DIAGNOSTIC    0x2D
150 #define EPS_APP_CRC     0x4B
```

```
151 #define EPS_RESTART     0x55
152 #define EPS_BW_SETTINGS   0x66
153 #define EPS_TCS_SETTINGS  0x78
154 #define EPS_PWR_SUB     0x87
155 #define EPS_RELEASE     0xAA
156 #define EPS_UPTIME      0xD2
157 #define EPS_CC_RST      0xE1
158
159 //EEPROM storage addresses
160 #define BW_SETTINGS_ADDR  0x00
161 #define TCS_SETTINGS_ADDR 0x04
162
163 // Watchdog_reset macro
164 #define WDT_Reset() asm("wdr")
165
166 // Delay macros
167 #define delay_ns(__ns) \
168     if((unsigned long) (F_CPU/1000000000.0 * __ns) != F_CPU/1000000000.0 *
        __ns)\
169           __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000000000.0 *
              __ns)+1);\
170     else __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000000000.0 *
        __ns))
171 #define delay_us(__us) \
172     if((unsigned long) (F_CPU/1000000.0 * __us) != F_CPU/1000000.0 * __us)\
173           __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000000.0 * __us
              )+1);\
174     else __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000000.0 * __us)
        )
175 #define delay_ms(__ms) \
176     if((unsigned long) (F_CPU/1000.0 * __ms) != F_CPU/1000.0 * __ms)\
177           __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000.0 * __ms)
              +1);\
178     else __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1000.0 * __ms))
179 #define delay_s(__s) \
180     if((unsigned long) (F_CPU/1.0 * __s) != F_CPU/1.0 * __s)\
181           __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1.0 * __s)+1);\
182     else __builtin_avr_delay_cycles((unsigned long) ( F_CPU/1.0 * __s))
183
184
185 // Declare global variables
186 extern TWI_Master_t twiMaster;  //TWI master module.
187 extern TWI_Slave_t twiSlave;  //TWI slave module.
188 extern TWI_Slave_t twiSlave2; //TWI secondary slave module
189 extern uint8_t ina226_i2c_adr_1[];
190 extern uint8_t ina226_i2c_adr_2[];
191 extern uint8_t ina226_i2c_adr_3[];
192 extern uint8_t tmp175_i2c_adr[];
193 extern uint8_t hamming_array[]; //Hamming encoding, distance 4
194 extern uint8_t comm_switch;
195
196 extern struct status{
197
```

```
198   register8_t EPS_OFF;  //Subsystem Offline
199   register8_t EPS_BW;   //Battery/Voltage Warning
200   register8_t EPS_NSE;  //New System Event
201   register8_t EPS_USE;  //Unhandled System Error/Warning
202
203 } statusEPS;
204
205 extern struct diagnostic{
206
207   //Status flags, max 13 byte
208   uint8_t packageID;
209   uint8_t epsID;
210   uint8_t sysEventFlag;
211   uint8_t restartFlag;
212   uint8_t sysOnline;
213   uint8_t enable_1_4;
214   uint8_t enable_5_8;
215   uint8_t fault_1_4;
216   uint8_t enable9_10;
217   uint8_t i2c_transaction_result; //default value 0xFF
218   uint8_t i2c_bus_state;      //default value 0xFF
219   uint8_t sensor_fault_1;
220   uint8_t sensor_fault_2;
221
222   //sensor data
223   int16_t vBat1; //15
224   int16_t vBat1_max;
225   int16_t vBat1_min;
226
227   int16_t vBat2;
228   int16_t vBat2_max;
229   int16_t vBat2_min;
230
231   int16_t vBat3;
232   int16_t vBat3_max;
233   int16_t vBat3_min;
234
235   int16_t vBat4; //33
236   uint16_t EMPTY_CRC_HOLDER1; //33 byte before and 62 byte after
237   int16_t vBat4_max;
238   int16_t vBat4_min;
239
240   int16_t cCHG1;
241   int16_t cCHG1_max;
242   int16_t cCHG1_min;
243
244   int16_t cCHG2;
245   int16_t cCHG2_max;
246   int16_t cCHG2_min; //51
247
248   int16_t cCHG3;
249   int16_t cCHG3_max;
250   int16_t cCHG3_min;
```

```
251
252   int16_t cCHG4;
253   int16_t cCHG4_max;
254   int16_t cCHG4_min;
255
256   int16_t batSoC; //65
257
258   int16_t cMainBus;
259   int16_t cMainBus_max;
260   int16_t cMainBus_min;
261
262   int16_t vCOMM;
263   int16_t vCOMM_max;
264   int16_t vCOMM_min;
265
266   int16_t vOBDH;
267   int16_t vOBDH_max;
268   int16_t vOBDH_min; //83
269
270   int16_t vADCS;
271   int16_t vADCS_max;
272   int16_t vADCS_min;
273
274   int16_t vPLD;
275   int16_t vPLD_max;
276   int16_t vPLD_min;
277
278   int16_t cCOMM; //97
279   uint16_t EMPTY_CRC_HOLDER2; //62 byte after
280   int16_t cCOMM_max;
281   int16_t cCOMM_min;
282
283   int16_t cOBDH;
284   int16_t cOBDH_max;
285   int16_t cOBDH_min; //109
286
287   int16_t cADCS;
288   int16_t cADCS_max;
289   int16_t cADCS_min;
290
291   int16_t cPLD;
292   int16_t cPLD_max;
293   int16_t cPLD_min;
294
295   int8_t tmpBat1;
296   int8_t tmpBat1_max;
297   int8_t tmpBat1_min; //124
298
299   int8_t tmpBat2;
300   int8_t tmpBat2_max;
301   int8_t tmpBat2_min;
302
303   int8_t tmpBat3;
```

```
304   int8_t tmpBat3_max;
305   int8_t tmpBat3_min;
306
307   int8_t tmpBackPanel;
308   int8_t tmpBackPanel_max;
309   int8_t tmpBackPanel_min; //133
310
311   uint8_t releaseStatus; //max 27 byte after
312   int16_t BW_set;
313   int16_t BW_clear;
314   uint8_t TCS_state;
315   int8_t TCS_enable;
316   int8_t TCS_disable;
317
318   uint16_t uptime_ticks;
319   uint8_t uptime_weeks; //144
320
321 } diagnosticEPS;
322
323 extern struct sysEvent{
324
325   uint8_t SENSOR_I2C_ERROR;
326   uint8_t SENSOR_OUT_OF_RANGE;
327   uint8_t UNKNOWN_STATE;
328   uint8_t UNKNOWN_CMD;
329
330 } sysEventEPS;
331
332 extern struct reset{
333
334   uint8_t rst_software;
335   uint8_t rst_watchdog;
336   uint8_t rst_powerOn;
337   uint8_t rst_external;
338
339 } resetEPS;
340
341 extern struct systems{
342
343   uint8_t OBDH_ON;
344   uint8_t PLD_ON;
345   uint8_t COMM_ON;
346   uint8_t ADC_ON;
347
348 } systemsOnline;
349
350 extern struct enable{
351
352   register8_t ENABLE1;
353   register8_t ENABLE2;
354   register8_t ENABLE3;
355   register8_t ENABLE4;
356   register8_t ENABLE5;
```

```
357   register8_t ENABLE6;
358   register8_t ENABLE7;
359   register8_t ENABLE8;
360
361 } enableEPS;
362
363 extern struct fault{
364
365   register8_t FAULT1;
366   register8_t FAULT2;
367   register8_t FAULT3;
368   register8_t FAULT4;
369
370 } faultEPS;
371
372 extern struct enable_9_10{
373
374   register8_t ENABLE9;
375   register8_t ENABLE10;
376   register8_t TCS_EN;
377   register8_t FAULT5;
378
379 } enableEPS_9_10;
380
381 extern struct sensorFault1{
382
383   uint8_t ina226_chg1;
384   uint8_t ina226_chg2;
385   uint8_t ina226_chg3;
386   uint8_t ina226_chg4;
387   uint8_t stc3100;
388   uint8_t ina226_mainbus;
389   uint8_t ina226_payload;
390   uint8_t ina226_adcs;
391
392 } sensorFault1EPS;
393
394 extern struct sensorFault2{
395
396   uint8_t ina226_obdh;
397   uint8_t ina226_comm;
398   uint8_t tmp175_bat1;
399   uint8_t tmp175_bat2;
400   uint8_t tmp175_bat3;
401   uint8_t tmp175_backpanel;
402   uint8_t dummy1;
403   uint8_t dummy2;
404
405 } sensorFault2EPS;
406
407 extern struct releaseDet{
408
409   uint8_t RELEASE_DET_1;
```

```
410   uint8_t RELEASE_DET_2;
411   uint8_t RELEASE_DET_3;
412   uint8_t RELEASE_DET_4;
413
414 } releaseDetEPS;
415
416 extern struct commands{
417
418   register8_t release;
419   register8_t releaseCMD;
420   register8_t rtc_ovf;
421   register8_t power;
422   register8_t powerMode;
423   register8_t powerSystem;
424   register8_t diagnosticCMD;
425   register8_t statusCMD;
426   register8_t BW_settings;
427   register8_t TCS_settings;
428   register8_t CC_reset;
429
430 } commandsEPS;
431
432 extern struct BWSettings{
433
434   union{
435     int16_t BW_set_16b;
436     struct{
437       int8_t BW_set_lsb;
438       int8_t BW_set_msb;
439     };
440   } BW_set;
441
442   union{
443     int16_t BW_clear_16b;
444     struct{
445       int8_t BW_clear_lsb;
446       int8_t BW_clear_msb;
447     };
448   } BW_clear;
449
450 } BWSettingsEPS;
451
452 extern struct TCSSettings{
453
454   uint8_t TCS_state;
455   int8_t TCS_enable;
456   int8_t TCS_disable;
457
458 } TCSSettingsEPS;
459
460 extern struct internalFlags{
461
462   register8_t i2c_transmission_error;
```

```
463   register8_t sensor_value_invalid;
464   register8_t bat_under_temp;
465   register8_t master_counter;
466   register8_t bw_alert;
467   register8_t bw_clear;
468   register8_t lowTemp;
469   register8_t normalTemp;
470   register8_t TCS_off;
471
472 } internalFlagsEPS;
473
474 extern struct uptime{
475
476   union{
477     uint16_t ticks_16b; //10 second tick counter
478     struct{
479       uint8_t ticks_lsb;
480       uint8_t ticks_msb;
481     };
482   } ticks;
483
484   uint8_t weeks; //week counter
485
486 } uptimeEPS;
487
488 #endif //EPS_H_
```

Listing E.2: EPS.c

```
1
2 /*
3  *  CubeSTAR EPS
4  *  Filename: EPS.c
5  *  Author: Knut Olav Skyttemyr
6  *  Description: main() and ISRs
7  */
8
9 // Includes
10 #include "EPS.h"
11
12 // Local prototypes
13 void start_init(void);
14 void start_conversion(void);
15 void fetch_sensordata(void);
16 void update_uptime(void);
17 void tps2557_fault(void);
18 void release_probes(uint8_t probe);
19 void i2c_timeout(void);
20 void check_rst(void);
21 void check_fault(void);
22
23 ISR(I2C_MASTER_vect){
24
25   TWI_MasterInterruptHandler(&twiMaster);
```

```
26 }
27
28 ISR(I2C_SLAVE_vect){
29
30   TWI_SlaveInterruptHandler(&twiSlave);
31 }
32
33 ISR(I2C_SLAVE2_vect){
34
35   TWI_SlaveInterruptHandler(&twiSlave2);
36 }
37
38 ISR(RTC_OVF_vect){
39
40   volatile static uint8_t count = 0;
41   volatile static uint8_t count2 = 0;
42
43   WDT_Reset();
44   if(++count == 4){
45     count = 0;
46     commandsEPS.rtc_ovf = 1; //set command flag
47   }
48
49   if(++internalFlagsEPS.master_counter == MASTER_TIMEOUT){
50
51     power_subsystems(1,0,1,0,TOGGLE); //power toggle OBDH and COMM (?) if
         regular status cmd not received
52     internalFlagsEPS.master_counter = 0;
53   }
54 }
55
56
57 ISR(FAULT_vect){
58
59   // Interrupt if rising or falling edge
60   tps2557_fault();
61 }
62
63
64 int main(void){
65
66   start_init();
67   load_eeprom();
68   check_rst();
69
70   //Wait until stable voltages before enabling the fault interrupt
71   delay_ms(500);
72   PORTCFG.MPCMASK = 0x1F; //pin 0-4 on portA.
73   PORTA.PIN0CTRL = PORT_ISC_FALLING_gc;
74   PORTA.INT0MASK = 0x1F; //pin 0-4 on portA.
75   PORTA.INTCTRL = PORT_INT1LVL_OFF_gc | PORT_INT0LVL_MED_gc; //interrupt
       level
76
```

```
77  // Sensor data header
78  #ifdef SENSOR
79    transmit_line((int8_t*) "\n\rCHG1[mV] CHG1[mA] CHG2[mV] CHG2[mA] CHG3[mV]
          CHG3[mA] CHG4[mV] CHG4[mA] SoC[mAh] Bat[mA] Bus[mA] COMM[mV] COMM[mA
        ] OBDH[mV] OBDH[mA] ADCS[mV] ADCS[mA] PLD[mV] PLD[mA] tmp(Bat1) tmp(
        Bat2) tmp(Bat3) tmp(uC)");
80  #endif
81
82    while(1){
83
84    // Sleep between interrupts
85    #ifdef SENSOR
86      delay_ms(10); //Allow UART to finish before going to sleep
87    #endif
88    sleep_enable();
89    while(RTC.STATUS & RTC_SYNCBUSY_bm);
90    sleep_cpu();
91    sleep_disable();
92
93    i2c_timeout(); //Allow ongoing I2C transmission to finish before
          continuing
94
95    if(commandsEPS.rtc_ovf == 1){
96
97      #ifdef SENSOR
98        NEWLINE;
99      #endif
100
101     start_conversion();
102     update_uptime();
103     fetch_sensordata();
104     check_fault();
105
106     // Clear flags and counters
107     internalFlagsEPS.bw_alert = 0;
108     internalFlagsEPS.bw_clear = 0;
109     internalFlagsEPS.lowTemp = 0;
110     internalFlagsEPS.normalTemp = 0;
111     commandsEPS.rtc_ovf = 0;
112   }
113
114   if(commandsEPS.release == 1){
115
116     // HD4 check
117     if(!HD4Valid(commandsEPS.releaseCMD)){
118
119       statusEPS.EPS_USE = 1; //Unhandled system error
120       sysEventEPS.UNKNOWN_CMD = 1;
121       #ifdef DEBUG
122         transmit_line((int8_t*) "\n\rEvent: Unknown command, probe release"
              );
123       #endif
124     }
```

```
125       else
126         release_probes(commandsEPS.releaseCMD);
127
128       commandsEPS.release = 0; //Clear flag
129     }
130
131     if(commandsEPS.power == 1){
132
133       // HD4 check
134       if(!HD4Valid(commandsEPS.powerSystem)){
135
136         statusEPS.EPS_USE = 1; //Unhandled system error
137         sysEventEPS.UNKNOWN_CMD = 1;
138         #ifdef DEBUG
139           transmit_line((int8_t*) "\n\rEvent: Unknown command, power
                subsystem");
140         #endif
141       }
142
143       if(commandsEPS.powerMode == 0xAA){
144
145         //Do nothing, only return status to OBDH
146       }
147
148       else if(commandsEPS.powerMode == 0xFF)
149         power_subsystems((0x10 & commandsEPS.powerSystem), (0x20 &
              commandsEPS.powerSystem), (0x40 & commandsEPS.powerSystem), (0x80
               & commandsEPS.powerSystem), ENABLE);
150
151       else if(commandsEPS.powerMode == 0x0F)
152         power_subsystems((0x10 & commandsEPS.powerSystem), (0x20 &
              commandsEPS.powerSystem), (0x40 & commandsEPS.powerSystem), (0x80
               & commandsEPS.powerSystem), TOGGLE);
153
154       else if(commandsEPS.powerMode == 0x00)
155         power_subsystems((0x10 & commandsEPS.powerSystem), (0x20 &
              commandsEPS.powerSystem), (0x40 & commandsEPS.powerSystem), (0x80
               & commandsEPS.powerSystem), DISABLE);
156
157       else{
158
159         sysEventEPS.UNKNOWN_CMD = 1; //Unknown command
160         statusEPS.EPS_USE = 1;
161         #ifdef DEBUG
162           transmit_line((int8_t*) "\n\rEvent: Unknown command, power mode");
163         #endif
164       }
165
166       commandsEPS.power = 0; //Clear flag
167     }
168
169     if(commandsEPS.diagnosticCMD == 1){
170
```

```
171       // Clear diagnostic flags
172       memset(&sysEventEPS.SENSOR_I2C_ERROR, 0, 4);
173       memset(&resetEPS.rst_software, 0, 4);
174       //systems online and enable flags are kept
175       diagnosticEPS.i2c_transaction_result = 0xFF;
176       diagnosticEPS.i2c_bus_state = 0xFF;
177       memset(&sensorFault1EPS.ina226_chg1, 0, 8);
178       memset(&sensorFault2EPS.ina226_obdh, 0, 8);
179       //release detection flags are kept
180
181       clear_statistics();
182       commandsEPS.diagnosticCMD = 0; //Clear flag
183
184       #ifdef DEBUG
185         transmit_line((int8_t*) "\n\rDiagnostic cmd");
186       #endif
187     }
188
189     if(commandsEPS.statusCMD){
190
191       #ifdef DEBUG
192         transmit_line((int8_t*) "\n\rStatus: ");
193         transmit_hex(HD4Value(&statusEPS.EPS_OFF));
194       #endif
195
196       //Clear USE and NSE flag
197       statusEPS.EPS_USE = 0;
198       statusEPS.EPS_NSE = 0;
199
200       internalFlagsEPS.master_counter = 0;
201       commandsEPS.statusCMD = 0;
202     }
203
204     if(commandsEPS.BW_settings == 1){
205
206       eeprom_write_block(&BWSettingsEPS.BW_set.BW_set_lsb, (uint8_t*)
              BW_SETTINGS_ADDR, 4);
207       eeprom_busy_wait(); //Wait until eeprom_write is finished
208       commandsEPS.BW_settings = 0; //Clear flag
209       #ifdef DEBUG
210         transmit_line((int8_t*)"\n\rNew BW-settings registered");
211       #endif
212     }
213
214     if(commandsEPS.TCS_settings == 1){
215
216       if(TCSSettingsEPS.TCS_state == 0)
217         internalFlagsEPS.TCS_off = 1;
218       eeprom_write_block(&TCSSettingsEPS.TCS_state, (uint8_t*)
              TCS_SETTINGS_ADDR, 3);
219       eeprom_busy_wait(); //Wait until eeprom_write is finished
220       commandsEPS.TCS_settings = 0; //Clear flag
221       #ifdef DEBUG
```

```
222         transmit_line((int8_t*)"\n\rNew TCS-settings registered");
223       #endif
224     }
225
226     if(commandsEPS.CC_reset == 1){
227
228       //Reset coulomb counter
229       for(uint8_t i=0; i<NR_OF_STC3100; i++){
230
231         if(!stc3100_reset(STC3100)){
232
233           #ifdef DEBUG
234             transmit_line((int8_t*) "\n\rError: I2C STC3100\t");
235           #endif
236
237           internalFlagsEPS.i2c_transmission_error = 1;
238           sysEventEPS.SENSOR_I2C_ERROR = 1;
239
240           error_handling();
241         }
242       }
243       commandsEPS.CC_reset = 0; //Clear flag
244     }
245     }
246 }
247
248
249 void update_uptime(void){
250
251   if(++uptimeEPS.ticks.ticks_16b >= 60480){ //24*7*60*6 = 60480. One tick is
        10 seconds
252
253     uptimeEPS.ticks.ticks_16b = 0;
254     uptimeEPS.weeks++;
255   }
256 }
257
258
259 void tps2557_fault(void){
260
261   //FAULT_1
262   if(((PORTA.IN & PIN0_bm) == 0) && (faultEPS.FAULT1 == 0) && (enableEPS.
        ENABLE1 == 1)){
263     faultEPS.FAULT1 = 1;
264     statusEPS.EPS_NSE = 1; //Raise new system event flag
265     #ifdef DEBUG
266       transmit_line((int8_t*) "\n\rAlert: Fault_1 detected");
267     #endif
268   }
269
270   //FAULT_2
271   if(((PORTA.IN & PIN1_bm) == 0) && (faultEPS.FAULT2 == 0) && (enableEPS.
        ENABLE2 == 1)){
```

```
272     faultEPS.FAULT2 = 1;
273     statusEPS.EPS_NSE = 1; //Raise new system event flag
274     #ifdef DEBUG
275       transmit_line((int8_t*) "\n\rAlert: Fault_2 detected");
276     #endif
277   }
278
279   //FAULT_3
280   if(((PORTA.IN & PIN2_bm) == 0) && (faultEPS.FAULT3 == 0) && (enableEPS.
        ENABLE3 == 1)){
281     faultEPS.FAULT3 = 1;
282     statusEPS.EPS_NSE = 1; //Raise new system event flag
283     #ifdef DEBUG
284       transmit_line((int8_t*) "\n\rAlert: Fault_3 detected");
285     #endif
286   }
287
288   //FAULT_4
289   if(((PORTA.IN & PIN3_bm) == 0) && (faultEPS.FAULT4 == 0) && (enableEPS.
        ENABLE4 == 1)){
290     faultEPS.FAULT4 = 1;
291     statusEPS.EPS_NSE = 1; //Raise new system event flag
292     #ifdef DEBUG
293       transmit_line((int8_t*) "\n\rAlert: Fault_4 detected");
294     #endif
295   }
296
297   //FAULT_5
298   if(((PORTA.IN & PIN4_bm) == 0) && (enableEPS_9_10.FAULT5 == 0) && (
        enableEPS.ENABLE5 == 1)){
299     enableEPS_9_10.FAULT5 = 1;
300     statusEPS.EPS_NSE = 1; //Raise new system event flag
301     #ifdef DEBUG
302       transmit_line((int8_t*) "\n\rAlert: Fault_5 detected");
303     #endif
304   }
305 }
306
307
308 void release_probes(uint8_t probe){
309
310   uint8_t count = 0;
311
312   if(probe == 0x00){
313
314     // Do nothing, only return of release status flag to OBDH.
315   }
316
317   else if(0x10 & probe){
318
319     // Check if probe is released already
320     if((PORTE.IN & PIN2_bm) == 0){
321
```

```
322       #ifdef DEBUG
323         transmit_line((int8_t*)"\n\rEvent: Release probe 1 circuit enabled");
324       #endif
325
326       PORTJ.OUTSET = (PIN2_bm | PIN3_bm); //RELEASE_CTRL1: EN6, PJ2-3
327       enableEPS.ENABLE6 = 1;
328
329       // Wait until release detected or time-out after 5 seconds (20ms*250=5s
              )
330       for(count = 0; count <= RELEASE_TIMEOUT; count++){
331         if((PORTE.IN & PIN2_bm) == PIN2_bm)
332           break;
333         else
334           delay_ms(20);
335       }
336
337       PORTJ.OUTCLR = (PIN2_bm | PIN3_bm);
338       enableEPS.ENABLE6 = 0;
339       #ifdef DEBUG
340         transmit_line((int8_t*)"\n\rEvent: Release probe 1 circuit disabled")
              ;
341       #endif
342     }
343
344     // Update flag
345     if((PORTE.IN & PIN2_bm) == PIN2_bm){
346       releaseDetEPS.RELEASE_DET_1 = 1;
347       #ifdef DEBUG
348         transmit_line((int8_t*)"\n\rEvent: Probe 1 successfully released");
349       #endif
350     }
351   }
352
353   else if(0x20 & probe){
354
355     // Check if probe is released already
356     if((PORTE.IN & PIN3_bm) == 0){
357
358       #ifdef DEBUG
359         transmit_line((int8_t*)"\n\rEvent: Release probe 2 circuit enabled");
360       #endif
361
362       PORTJ.OUTSET = (PIN4_bm | PIN5_bm); //RELEASE_CTRL2: EN7, PJ4-5
363       enableEPS.ENABLE7 = 1;
364
365       for(count = 0; count <= RELEASE_TIMEOUT; count++){
366         if((PORTE.IN & PIN3_bm) == PIN3_bm)
367           break;
368         else
369           delay_ms(20);
370       }
371
372       PORTJ.OUTCLR = (PIN4_bm | PIN5_bm);
```

```
373       enableEPS.ENABLE7 = 0;
374       #ifdef DEBUG
375         transmit_line((int8_t*)"\n\rEvent: Release probe 2 circuit disabled")
              ;
376       #endif
377     }
378
379     if((PORTE.IN & PIN3_bm) == PIN3_bm){
380       releaseDetEPS.RELEASE_DET_2 = 1;
381       #ifdef DEBUG
382         transmit_line((int8_t*)"\n\rEvent: Probe 2 successfully released");
383       #endif
384     }
385   }
386
387   else if(0x40 & probe){
388
389     // Check if probe is released already
390     if((PORTE.IN & PIN4_bm) == 0){
391
392       #ifdef DEBUG
393         transmit_line((int8_t*)"\n\rEvent: Release probe 3 circuit enabled");
394       #endif
395
396       PORTJ.OUTSET = (PIN6_bm | PIN7_bm); //RELEASE_CTRL3: EN8, PJ6-7
397       enableEPS.ENABLE8 = 1;
398
399       for(count = 0; count <= RELEASE_TIMEOUT; count++){
400         if((PORTE.IN & PIN4_bm) == PIN4_bm)
401           break;
402         else
403           delay_ms(20);
404       }
405
406       PORTJ.OUTCLR = (PIN6_bm | PIN7_bm);
407       enableEPS.ENABLE8 = 0;
408       #ifdef DEBUG
409         transmit_line((int8_t*)"\n\rEvent: Release probe 3 circuit disabled")
              ;
410       #endif
411     }
412
413     if((PORTE.IN & PIN4_bm) == PIN4_bm){
414       releaseDetEPS.RELEASE_DET_3 = 1;
415       #ifdef DEBUG
416         transmit_line((int8_t*)"\n\rEvent: Probe 3 successfully released");
417       #endif
418     }
419   }
420
421   else if(0x80 & probe){
422
423     // Check if probe is released already
```

```
424     if((PORTE.IN & PIN5_bm) == 0){
425
426       #ifdef DEBUG
427         transmit_line((int8_t*)"\n\rEvent: Release probe 4 circuit enabled");
428       #endif
429
430       PORTK.OUTSET = (PIN0_bm | PIN1_bm); //RELEASE_CTRL4: EN9, PK0-1
431       enableEPS_9_10.ENABLE9 = 1;
432
433       for(count = 0; count <= RELEASE_TIMEOUT; count++){
434         if((PORTE.IN & PIN5_bm) == PIN5_bm)
435           break;
436         else
437           delay_ms(20);
438       }
439
440       PORTK.OUTCLR = (PIN0_bm | PIN1_bm);
441       enableEPS_9_10.ENABLE9 = 0;
442       #ifdef DEBUG
443         transmit_line((int8_t*)"\n\rEvent: Release probe 4 circuit disabled")
              ;
444       #endif
445     }
446
447     if((PORTE.IN & PIN5_bm) == PIN5_bm){
448       releaseDetEPS.RELEASE_DET_4 = 1;
449       #ifdef DEBUG
450         transmit_line((int8_t*)"\n\rEvent: Probe 4 successfully released");
451       #endif
452     }
453   }
454
455   else{
456
457     statusEPS.EPS_USE = 1; //Unhandled system error
458     sysEventEPS.UNKNOWN_CMD = 1;
459     #ifdef DEBUG
460       transmit_line((int8_t*)"\n\rError: Unknown command");
461     #endif
462   }
463 }
464
465
466 void power_subsystems(uint8_t obdh, uint8_t pld, uint8_t comm, uint8_t adcs,
      uint8_t mode){
467
468   // OBDH
469   if(obdh && (mode == ENABLE)){
470     enableEPS.ENABLE3 = 1;
471     PORTH.OUTSET = (PIN4_bm | PIN5_bm); //OBDH: EN3, PH4-5
472     PORTF.OUTSET = PIN4_bm; //nCS_CN5: PF4
473     systemsOnline.OBDH_ON = 1;
474     #ifdef DEBUG
```

```
475       transmit_line((int8_t*) "\n\rEvent: OBDH enabled");
476     #endif
477   }
478   else if(obdh && (mode == TOGGLE)){
479     enableEPS.ENABLE3 = 0;
480     PORTH.OUTCLR = (PIN4_bm | PIN5_bm); //OBDH: EN3, PH4-5
481     PORTF.OUTCLR = PIN4_bm; //nCS_CN5: PF4
482     delay_ms(POWER_TOGGLE_DLY); //Delay until capacitors are discharged
483     PORTH.OUTSET = (PIN4_bm | PIN5_bm); //OBDH: EN3, PH4-5
484     enableEPS.ENABLE3 = 1;
485     PORTF.OUTSET = PIN4_bm; //nCS_CN5: PF4
486     systemsOnline.OBDH_ON = 1;
487     #ifdef DEBUG
488       transmit_line((int8_t*) "\n\rEvent: OBDH power toggle");
489     #endif
490   }
491   else if(obdh && (mode == DISABLE)){
492     enableEPS.ENABLE3 = 0;
493     PORTH.OUTCLR = (PIN4_bm | PIN5_bm); //OBDH: EN3, PH4-5
494     PORTF.OUTCLR = PIN4_bm; //nCS_CN5: PF4
495     systemsOnline.OBDH_ON = 0;
496     #ifdef DEBUG
497       transmit_line((int8_t*) "\n\rEvent: OBDH disabled");
498     #endif
499   }
500   //else
501     //do nothing
502
503   // Payload
504   if(pld && (mode == ENABLE)){
505     enableEPS.ENABLE1 = 1;
506     PORTH.OUTSET = (PIN0_bm | PIN1_bm); //PLD: EN1, PH0-1
507     PORTF.OUTSET = PIN0_bm; //nCS_CN1: PF0
508     systemsOnline.PLD_ON = 1;
509     #ifdef DEBUG
510       transmit_line((int8_t*) "\n\rEvent: PLD enabled");
511     #endif
512   }
513   else if(pld && (mode == TOGGLE)){
514     enableEPS.ENABLE1 = 0;
515     PORTH.OUTCLR = (PIN0_bm | PIN1_bm); //PLD: EN1, PH0-1
516     PORTF.OUTCLR = PIN0_bm; //nCS_CN1: PF0
517     delay_ms(POWER_TOGGLE_DLY); //Delay until capacitors are discharged
518     PORTH.OUTSET = (PIN0_bm | PIN1_bm); //PLD: EN1, PH0-1
519     enableEPS.ENABLE1 = 1;
520     PORTF.OUTSET = PIN0_bm; //nCS_CN1: PF0
521     systemsOnline.PLD_ON = 1;
522     #ifdef DEBUG
523       transmit_line((int8_t*) "\n\rEvent: PLD power toggle");
524     #endif
525   }
526   else if(pld && (mode == DISABLE)){
527     enableEPS.ENABLE1 = 0;
```

```
528     PORTH.OUTCLR = (PIN0_bm | PIN1_bm); //PLD: EN1, PH0-1
529     PORTF.OUTCLR = PIN0_bm; //nCS_CN1: PF0
530     systemsOnline.PLD_ON = 0;
531     #ifdef DEBUG
532       transmit_line((int8_t*) "\n\rEvent: PLD disabled");
533     #endif
534   }
535   //else
536     //do nothing
537
538   // COMM
539   if(comm && (mode == ENABLE)){
540
541     if(comm_switch == 0){
542       enableEPS.ENABLE5 = 0;
543       enableEPS.ENABLE4 = 1;
544       PORTJ.OUTCLR = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
545       PORTH.OUTSET = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
546       PORTF.OUTSET = PIN5_bm; //nCS_CN6: PF5
547       systemsOnline.COMM_ON = 1;
548       #ifdef DEBUG
549         transmit_line((int8_t*) "\n\rEvent: COMM_0 enabled");
550       #endif
551     }
552     else if(comm_switch == 1){
553       enableEPS.ENABLE4 = 0;
554       enableEPS.ENABLE5 = 1;
555       PORTH.OUTCLR = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
556       PORTJ.OUTSET = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
557       PORTF.OUTSET = PIN5_bm; //nCS_CN6: PF5
558       systemsOnline.COMM_ON = 1;
559       #ifdef DEBUG
560         transmit_line((int8_t*) "\n\rEvent: COMM_1 enabled");
561       #endif
562     }
563     else {
564
565       statusEPS.EPS_USE = 1; //Unhandled system error
566       sysEventEPS.UNKNOWN_STATE = 1;
567       #ifdef DEBUG
568         transmit_line((int8_t*)"\n\rError: Unknown state, Comm enable");
569       #endif
570     }
571
572   }
573   else if(comm && (mode == TOGGLE)){
574
575     if(comm_switch == 0){
576       enableEPS.ENABLE4 = 0;
577       enableEPS.ENABLE5 = 0;
578       PORTH.OUTCLR = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
579       PORTJ.OUTCLR = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
580       PORTF.OUTCLR = PIN5_bm; //nCS_CN6: PF5
```

```
581       delay_ms(POWER_TOGGLE_DLY); //Delay until capacitors are discharged
582       PORTH.OUTSET = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7 & PJ0-1
583       enableEPS.ENABLE4 = 1;
584       PORTF.OUTSET = PIN5_bm; //nCS_CN6: PF5
585       systemsOnline.COMM_ON = 1;
586       #ifdef DEBUG
587         transmit_line((int8_t*) "\n\rEvent: COMM_0 power toggle");
588       #endif
589     }
590     else if(comm_switch == 1){
591       enableEPS.ENABLE5 = 0;
592       enableEPS.ENABLE4 = 0;
593       PORTJ.OUTCLR = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
594       PORTH.OUTCLR = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
595       PORTF.OUTCLR = PIN5_bm; //nCS_CN6: PF5
596       delay_ms(POWER_TOGGLE_DLY); //Delay until capacitors are discharged
597       PORTJ.OUTSET = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
598       enableEPS.ENABLE5 = 1;
599       PORTF.OUTSET = PIN5_bm; //nCS_CN6: PF5
600       enableEPS.ENABLE5 = 1;
601       systemsOnline.COMM_ON = 1;
602       #ifdef DEBUG
603         transmit_line((int8_t*) "\n\rEvent: COMM_1 power toggle");
604       #endif
605     }
606     else {
607
608       statusEPS.EPS_USE = 1; //Unhandled system error
609       sysEventEPS.UNKNOWN_STATE = 1;
610       #ifdef DEBUG
611         transmit_line((int8_t*)"\n\rError: Unknown state, Comm toggle");
612       #endif
613     }
614   }
615   else if(comm && (mode == DISABLE)){
616
617     if(comm_switch == 0){
618       enableEPS.ENABLE4 = 0;
619       enableEPS.ENABLE5 = 0;
620       PORTH.OUTCLR = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
621       PORTJ.OUTCLR = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
622       PORTF.OUTCLR = PIN5_bm; //nCS_CN6: PF5
623       systemsOnline.COMM_ON = 0;
624       #ifdef DEBUG
625         transmit_line((int8_t*) "\n\rEvent: COMM_0 disabled");
626       #endif
627     }
628     else if(comm_switch == 1){
629       enableEPS.ENABLE5 = 0;
630       enableEPS.ENABLE4 = 0;
631       PORTJ.OUTCLR = (PIN0_bm | PIN1_bm); //COMM: EN5, PJ0-1
632       PORTH.OUTCLR = (PIN6_bm | PIN7_bm); //COMM: EN4, PH6-7
633       PORTF.OUTCLR = PIN5_bm; //nCS_CN6: PF5
```

```
634       systemsOnline.COMM_ON = 0;
635       #ifdef DEBUG
636         transmit_line((int8_t*) "\n\rEvent: COMM_1 disabled");
637       #endif
638     }
639     else {
640
641       statusEPS.EPS_USE = 1; //Unhandled system error
642       sysEventEPS.UNKNOWN_STATE = 1;
643       #ifdef DEBUG
644         transmit_line((int8_t*)"\n\rError: Unknown state, Comm disable");
645       #endif
646     }
647   }
648   //else
649     //do nothing
650
651   // ADCS
652   if(adcs && (mode == ENABLE)){
653     enableEPS.ENABLE2 = 1;
654     PORTH.OUTSET = (PIN2_bm | PIN3_bm); //ADCS: EN2, PH2-3
655     PORTF.OUTSET = PIN1_bm; //nCS_CN2: PF1
656     systemsOnline.ADC_ON = 1;
657     #ifdef DEBUG
658       transmit_line((int8_t*) "\n\rEvent: ADCS enabled");
659     #endif
660   }
661   else if(adcs && (mode == TOGGLE)){
662     enableEPS.ENABLE2 = 0;
663     PORTH.OUTCLR = (PIN2_bm | PIN3_bm); //ADCS: EN2, PH2-3
664     PORTF.OUTCLR = PIN1_bm; //nCS_CN2: PF1
665     delay_ms(POWER_TOGGLE_DLY); //Delay until capacitors are discharged
666     PORTH.OUTSET = (PIN2_bm | PIN3_bm); //ADCS: EN2, PH2-3
667     enableEPS.ENABLE2 = 1;
668     PORTF.OUTSET = PIN1_bm; //nCS_CN2: PF1
669     systemsOnline.ADC_ON = 1;
670     #ifdef DEBUG
671       transmit_line((int8_t*) "\n\rEvent: ADCS power toggle");
672     #endif
673   }
674   else if(adcs && (mode == DISABLE)){
675     enableEPS.ENABLE2 = 0;
676     PORTH.OUTCLR = (PIN2_bm | PIN3_bm); //ADCS: EN2, PH2-3
677     PORTF.OUTCLR = PIN1_bm; //nCS_CN2: PF1
678     systemsOnline.ADC_ON = 0;
679     #ifdef DEBUG
680       transmit_line((int8_t*) "\n\rEvent: ADCS disabled");
681     #endif
682   }
683   //else
684     //do nothing
685
```

```
686   if((systemsOnline.ADC_ON == 0) || (systemsOnline.COMM_ON == 0) || (
          systemsOnline.OBDH_ON == 0) || (systemsOnline.PLD_ON == 0))
687     statusEPS.EPS_OFF = 1; //Set subsystem off flag
688   else
689     statusEPS.EPS_OFF = 0; //Clear subsystem off flag
690 }
691
692
693 void i2c_timeout(void){
694
695   if(twiSlave.status == TWIS_STATUS_BUSY){
696
697     for(uint16_t i=0; i<500; i++){
698
699       if(twiSlave.status == TWIS_STATUS_READY)
700         break;
701       delay_us(100);
702     }
703   }
704
705   if(twiSlave2.status == TWIS_STATUS_BUSY){
706
707     for(uint16_t i=0; i<500; i++){
708
709       if(twiSlave2.status == TWIS_STATUS_READY)
710         break;
711       delay_us(100);
712     }
713   }
714 }
715
716
717 void clear_statistics(void){
718
719   //0x8000 = min value, 0x7FFF = max value
720
721   diagnosticEPS.vBat1_max = 0x8000;
722   diagnosticEPS.vBat1_min = 0x7FFF;
723   diagnosticEPS.vBat2_max = 0x8000;
724   diagnosticEPS.vBat2_min = 0x7FFF;
725   diagnosticEPS.vBat3_max = 0x8000;
726   diagnosticEPS.vBat3_min = 0x7FFF;
727   diagnosticEPS.vBat4_max = 0x8000;
728   diagnosticEPS.vBat4_min = 0x7FFF;
729
730   diagnosticEPS.cCHG1_max = 0x8000;
731   diagnosticEPS.cCHG1_min = 0x7FFF;
732   diagnosticEPS.cCHG2_max = 0x8000;
733   diagnosticEPS.cCHG2_min = 0x7FFF;
734   diagnosticEPS.cCHG3_max = 0x8000;
735   diagnosticEPS.cCHG3_min = 0x7FFF;
736   diagnosticEPS.cCHG4_max = 0x8000;
737   diagnosticEPS.cCHG4_min = 0x7FFF;
```

```
738
739   diagnosticEPS.cMainBus_max = 0x8000;
740   diagnosticEPS.cMainBus_min = 0x7FFF;
741
742   diagnosticEPS.vCOMM_max = 0x8000;
743   diagnosticEPS.vCOMM_min = 0x7FFF;
744   diagnosticEPS.vOBDH_max = 0x8000;
745   diagnosticEPS.vOBDH_min = 0x7FFF;
746   diagnosticEPS.vADCS_max = 0x8000;
747   diagnosticEPS.vADCS_min = 0x7FFF;
748   diagnosticEPS.vPLD_max = 0x8000;
749   diagnosticEPS.vPLD_min = 0x7FFF;
750
751   diagnosticEPS.cCOMM_max = 0x8000;
752   diagnosticEPS.cCOMM_min = 0x7FFF;
753   diagnosticEPS.cOBDH_max = 0x8000;
754   diagnosticEPS.cOBDH_min = 0x7FFF;
755   diagnosticEPS.cADCS_max = 0x8000;
756   diagnosticEPS.cADCS_min = 0x7FFF;
757   diagnosticEPS.cPLD_max = 0x8000;
758   diagnosticEPS.cPLD_min = 0x7FFF;
759
760   diagnosticEPS.tmpBat1_max = 0x80;
761   diagnosticEPS.tmpBat1_min = 0x7F;
762   diagnosticEPS.tmpBat2_max = 0x80;
763   diagnosticEPS.tmpBat2_min = 0x7F;
764   diagnosticEPS.tmpBat3_max = 0x80;
765   diagnosticEPS.tmpBat3_min = 0x7F;
766   diagnosticEPS.tmpBackPanel_max = 0x80;
767   diagnosticEPS.tmpBackPanel_min = 0x7F;
768 }
769
770
771 void check_rst(void){
772
773   if((RST.STATUS & RST_SRF_bm) == RST_SRF_bm){
774
775     resetEPS.rst_software = 1;
776     statusEPS.EPS_NSE = 1;
777
778     #ifdef DEBUG
779       transmit_line((int8_t*) "\n\rReset: Software");
780     #endif
781   }
782
783   if((RST.STATUS & RST_WDRF_bm) == RST_WDRF_bm){
784
785     resetEPS.rst_watchdog = 1;
786     statusEPS.EPS_NSE = 1;
787
788     #ifdef DEBUG
789       transmit_line((int8_t*) "\n\rReset: Watchdog");
790     #endif
```

```
791   }
792
793   if((RST.STATUS & RST_PORF_bm) == RST_PORF_bm){
794
795     resetEPS.rst_powerOn = 1;
796     statusEPS.EPS_NSE = 1;
797
798     #ifdef DEBUG
799       transmit_line((int8_t*) "\n\rReset: Power-on");
800     #endif
801   }
802
803   if((RST.STATUS & RST_EXTRF_bm) == RST_EXTRF_bm){
804
805     resetEPS.rst_external = 1;
806     statusEPS.EPS_NSE = 1;
807
808     #ifdef DEBUG
809       transmit_line((int8_t*) "\n\rReset: External");
810     #endif
811   }
812
813   RST.STATUS = 0x3F; //Clear flags after reading
814 }
815
816
817 void load_eeprom(void){
818
819   eeprom_read_block(&BWSettingsEPS.BW_set.BW_set_lsb, (uint8_t*)
          BW_SETTINGS_ADDR, 4);
820   // If EEPROM is empty, load default values
821   if((BWSettingsEPS.BW_set.BW_set_16b == 0xFFFF) && (BWSettingsEPS.BW_clear.
          BW_clear_16b == 0xFFFF)){
822
823     BWSettingsEPS.BW_set.BW_set_16b = V_BAT_WARNING;
824     BWSettingsEPS.BW_clear.BW_clear_16b = V_BAT_SAFE;
825     #ifdef DEBUG
826       transmit_line((int8_t*)"\n\rLoading default BW settings...");
827     #endif
828   }
829
830   eeprom_read_block(&TCSSettingsEPS.TCS_state, (uint8_t*) TCS_SETTINGS_ADDR,
          3);
831   // If EEPROM is empty, load default values
832   if((TCSSettingsEPS.TCS_state == 0xFF) && (TCSSettingsEPS.TCS_enable == (
          int8_t)0xFF) && (TCSSettingsEPS.TCS_disable == (int8_t)0xFF)){
833
834     TCSSettingsEPS.TCS_state = TCS_DEFAULT;
835     TCSSettingsEPS.TCS_enable = TCS_ENABLE;
836     TCSSettingsEPS.TCS_disable = TCS_DISABLE;
837     #ifdef DEBUG
838       transmit_line((int8_t*)"\n\rLoading default TCS settings...");
839     #endif
```

```
840   }
841 }
842
843
844 void check_fault(void){
845
846   if(((PORTA.IN & PIN0_bm) == PIN0_bm) && (faultEPS.FAULT1 == 1)){
847     faultEPS.FAULT1 = 0;
848     statusEPS.EPS_NSE = 1; //Raise new system event flag
849     #ifdef DEBUG
850       transmit_line((int8_t*) "\n\rAlert: Fault_1 cleared");
851     #endif
852   }
853
854   if(((PORTA.IN & PIN1_bm) == PIN1_bm) && (faultEPS.FAULT2 == 1)){
855     faultEPS.FAULT2 = 0;
856     statusEPS.EPS_NSE = 1; //Raise new system event flag
857     #ifdef DEBUG
858       transmit_line((int8_t*) "\n\rAlert: Fault_2 cleared");
859     #endif
860   }
861
862   if(((PORTA.IN & PIN2_bm) == PIN2_bm) && (faultEPS.FAULT3 == 1)){
863     faultEPS.FAULT3 = 0;
864     statusEPS.EPS_NSE = 1; //Raise new system event flag
865     #ifdef DEBUG
866       transmit_line((int8_t*) "\n\rAlert: Fault_3 cleared");
867     #endif
868   }
869
870   if(((PORTA.IN & PIN3_bm) == PIN3_bm) && (faultEPS.FAULT4 == 1)){
871     faultEPS.FAULT4 = 0;
872     statusEPS.EPS_NSE = 1; //Raise new system event flag
873     #ifdef DEBUG
874       transmit_line((int8_t*) "\n\rAlert: Fault_4 cleared");
875     #endif
876   }
877
878   if(((PORTA.IN & PIN4_bm) == PIN4_bm) && (enableEPS_9_10.FAULT5 == 1)){
879     enableEPS_9_10.FAULT5 = 0;
880     statusEPS.EPS_NSE = 1; //Raise new system event flag
881     #ifdef DEBUG
882       transmit_line((int8_t*) "\n\rAlert: Fault_5 cleared");
883     #endif
884   }
885 }
```

Listing E.3: EPS_init.c

```
1
2 /*
3  *  CubeSTAR EPS
4  *  Filename: EPS_init.c
5  *  Author: Knut Olav Skyttemyr
```

```
 *  Description: Initialization of MCU and sensors
 */

// Includes
#include "EPS.h"

// Local prototypes
void init_system(void);
void init_pins(void);
void init_uart(void);
void init_i2c(void);
void startup(void);

//Define global variables
TWI_Master_t twiMaster;
TWI_Slave_t twiSlave;
TWI_Slave_t twiSlave2;
uint8_t ina226_i2c_adr_1[] = {INA226_CHG1, INA226_CHG2, INA226_CHG3,
    INA226_CHG4};
uint8_t ina226_i2c_adr_2[] = {INA226_BAT, INA226_MAINBUS};
uint8_t ina226_i2c_adr_3[] = {INA226_COMM, INA226_OBDH, INA226_ADCS,
    INA226_PLD};
uint8_t tmp175_i2c_adr[] = {TMP175_BAT1, TMP175_BAT2, TMP175_BAT3,
    TMP175_BACKPANEL};
uint8_t hamming_array[] = { 0x00, 0x1E, 0x2D, 0x33, 0x4B, 0x55, 0x66, 0x78,
              0x87, 0x99, 0xAA, 0xB4, 0xCC, 0xD2, 0xE1, 0xFF};
uint8_t comm_switch = 0;

struct status statusEPS;
struct diagnostic diagnosticEPS;
struct sysEvent sysEventEPS;
struct reset resetEPS;
struct systems systemsOnline;
struct enable enableEPS;
struct fault faultEPS;
struct enable_9_10 enableEPS_9_10;
struct sensorFault1 sensorFault1EPS;
struct sensorFault2 sensorFault2EPS;
struct releaseDet releaseDetEPS;
struct commands commandsEPS;
struct BWSettings BWSettingsEPS;
struct TCSSettings TCSSettingsEPS;
struct internalFlags internalFlagsEPS;
struct uptime uptimeEPS;

void start_init(void){

  init_system();
  init_pins();
  #ifndef NORMAL
    init_uart();
  #endif
  init_i2c();
```

```
56   sei(); //Enable global interrupts
57   startup();
58 }
59
60
61 void init_system(void){
62
63   // Oscillators
64   OSC.PLLCTRL = OSC_PLLSRC_RC2M_gc | 0x6; //6x PLL
65   OSC.XOSCCTRL = OSC_XOSCSEL_32KHz_gc; //start external 32kHz osc
66   OSC.CTRL = OSC_PLLEN_bm | OSC_RC2MEN_bm | OSC_XOSCEN_bm;
67   while(!(OSC.STATUS & OSC_PLLRDY_bm)); //wait for PLL ready
68   while(!(OSC.STATUS & OSC_RC2MRDY_bm)); //wait for 2MHz osc ready
69   while(!(OSC.STATUS & OSC_XOSCRDY_bm)); //wait for external osc ready
70
71   // System clock
72   CCP = CCP_IOREG_gc; //enable writing to protected registers
73   CLK.CTRL = CLK_SCLKSEL_PLL_gc; //PLL
74   delay_us(10);
75
76   // Real-time clock
77   CLK.RTCCTRL = CLK_RTCSRC_TOSC_gc | CLK_RTCEN_bm; //RTC: 1024Hz from 32768Hz
         external osc
78   delay_us(10);
79   RTC.CTRL = RTC_PRESCALER_DIV1_gc; //1024Hz real-time clock
80   RTC.INTCTRL = RTC_COMPINTLVL_OFF_gc | RTC_OVFINTLVL_HI_gc; //priority level
         interrupts.
81   while(RTC.STATUS & RTC_SYNCBUSY_bm);
82   RTC.PER = (uint16_t) (256*OVF_SEC); //overflow interrupt every x/4 second
83   while(RTC.STATUS & RTC_SYNCBUSY_bm);
84   RTC.COMP = 0;
85
86   // Auto-calibration
87   OSC.DFLLCTRL = OSC_RC2MCREF_bm; //2MHz clock calibration from external osc
88   DFLLRC2M.CTRL = DFLL_ENABLE_bm;
89
90   // Watch-dog timer
91   CCP = CCP_IOREG_gc; //enable writing to protected registers
92   WDT.CTRL = WDT_PER_8KCLK_gc | WDT_ENABLE_bm | WDT_CEN_bm; //8.0 sec time-
       out period
93
94   // Interrupts
95   PMIC.CTRL = PMIC_HILVLEN_bm | PMIC_MEDLVLEN_bm | PMIC_LOLVLEN_bm; //enable
       high, medium and low level interrupts
96
97   // Power reduction
98   PR.PRGEN |= PR_AES_bm | PR_DMA_bm | PR_EVSYS_bm | PR_EBI_bm;
99
100  // Disable all peripherals not used to save power
101  PR.PRPA = PR_DAC_bm | PR_ADC_bm | PR_AC_bm;
102  PR.PRPB = PR_DAC_bm | PR_ADC_bm | PR_AC_bm;
103  PR.PRPC = PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm | PR_TC1_bm
         | PR_TC0_bm;
```

```
104   PR.PRPD = PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm | PR_TC1_bm
          | PR_TC0_bm;
105   PR.PRPE = PR_USART1_bm | PR_USART0_bm | PR_SPI_bm | PR_HIRES_bm | PR_TC1_bm
          | PR_TC0_bm;
106   #ifdef NORMAL
107     PR.PRPF = PR_TWI_bm | PR_USART1_bm | PR_USART0_bm | PR_SPI_bm |
            PR_HIRES_bm | PR_TC1_bm | PR_TC0_bm;
108   #else
109     PR.PRPF = PR_TWI_bm | PR_USART1_bm | PR_SPI_bm | PR_HIRES_bm | PR_TC1_bm
            | PR_TC0_bm;
110   #endif
111
112   // Sleep mode
113   set_sleep_mode(SLEEP_SMODE_ESTDBY_gc); //use extended standby for faster
          startup, power-save for lower power usage.
114 }
115
116
117 void init_pins(void){
118
119   // Fault signals
120   PORTA.DIR = 0x00; //set pins to input
121
122   // EPS IRQ
123   PORTB.DIR = 0x04; //pin2 output
124
125   // Release detection signals
126   PORTE.DIR = 0x00; //set pins to input
127   PORTCFG.MPCMASK = 0x3C; //pin 2-5 on portE.
128   PORTE.PIN2CTRL =  PORT_OPC_PULLUP_gc | PORT_INVEN_bm;
129
130   // Enable signals
131   PORTH.DIR = 0xFF; //set pins to output
132   PORTJ.DIR = 0xFF; //set pins to output
133   PORTK.DIR = 0x3F; //set pin 0-5 to output
134
135   // nCS signals
136   PORTF.DIR = 0x73; //set pin 0-1,4-6 to output
137   PORTCFG.MPCMASK = 0x73;
138   PORTF.PIN0CTRL = PORT_INVEN_bm; //nCS is active low, invert signals.
139
140   // Pull-up on all unused pins for power reduction
141   PORTCFG.MPCMASK = 0xE0; //pin 5-7 on PortA
142   PORTA.PIN5CTRL = PORT_OPC_PULLUP_gc;
143   PORTCFG.MPCMASK = 0x0B; //pin 0-1,3 on PortB
144   PORTB.PIN3CTRL = PORT_OPC_PULLUP_gc;
145   PORTCFG.MPCMASK = 0xFC; //pin 2-7 on PortC
146   PORTC.PIN2CTRL = PORT_OPC_PULLUP_gc;
147   PORTCFG.MPCMASK = 0xFC; //pin 2-7 on PortD
148   PORTD.PIN2CTRL = PORT_OPC_PULLUP_gc;
149   PORTCFG.MPCMASK = 0xC0; //pin 6-7 on PortE
150   PORTE.PIN6CTRL = PORT_OPC_PULLUP_gc;
151   PORTCFG.MPCMASK = 0xC0; //pin 6-7 on PortK
```

```
152   PORTK.PIN6CTRL = PORT_OPC_PULLUP_gc;
153   PORTCFG.MPCMASK = 0x0C; //pin 2-3 on PortQ
154   PORTQ.PIN2CTRL = PORT_OPC_PULLUP_gc;
155   PORTCFG.MPCMASK = 0x03; //pin 0-1 on PortR
156   PORTR.PIN0CTRL = PORT_OPC_PULLUP_gc;
157
158   #ifdef NORMAL
159     PORTCFG.MPCMASK = 0x8C; //Pin2-3,7 on portF
160     PORTF.PIN0CTRL = PORT_OPC_PULLUP_gc; //when not using UART
161   #else
162     PORTF.PIN7CTRL = PORT_OPC_PULLUP_gc; //when using UART
163   #endif
164 }
165
166
167 void init_uart(void){
168
169   UART.CTRLA = USART_RXCINTLVL_OFF_gc | USART_TXCINTLVL_OFF_gc |
        USART_DREINTLVL_OFF_gc; //USART interrupts
170   UART.CTRLC = USART_CMODE_ASYNCHRONOUS_gc | USART_PMODE_DISABLED_gc |
        USART_CHSIZE_8BIT_gc; //asynk, no parity, 1 stop bit, 8bit data
171   UART.BAUDCTRLA = (uint8_t) BSEL_VALUE;
172   UART.BAUDCTRLB = (BSCALE_VALUE << 4) | (BSEL_VALUE >> 8);
173   UART.CTRLB = USART_TXEN_bm | USART_CLK2X_bm; //enable transmitting
174   UART_PORT.DIRSET = 0x08; //pin3 transmit is output
175 }
176
177
178 void init_i2c(void){
179
180   TWI_MasterInit(&twiMaster, &I2C_BUS4, TWI_MASTER_INTLVL_HI_gc,
        TWI_BAUDSETTING);
181   TWI_SlaveInitializeDriver(&twiSlave, &I2C_BUS1, TWI_SlaveProcessData);
182   TWI_SlaveInitializeModule(&twiSlave, EPS_I2C_ADDRESS,
        TWI_SLAVE_INTLVL_HI_gc);
183   TWI_SlaveInitializeDriver(&twiSlave2, &I2C_BUS2, TWI_SlaveProcessData);
184   TWI_SlaveInitializeModule(&twiSlave2, EPS_I2C_ADDRESS,
        TWI_SLAVE_INTLVL_HI_gc);
185 }
186
187 void startup(void){
188
189   #ifdef DEBUG
190     transmit_line((int8_t*) "\n\n\rStartup");
191   #endif
192
193   // Check probe release status at startup
194   if((PORTE.IN & PIN2_bm) == PIN2_bm)
195     releaseDetEPS.RELEASE_DET_1 = 1;
196   if((PORTE.IN & PIN3_bm) == PIN3_bm)
197     releaseDetEPS.RELEASE_DET_2 = 1;
198   if((PORTE.IN & PIN4_bm) == PIN4_bm)
199     releaseDetEPS.RELEASE_DET_3 = 1;
```

```
200   if((PORTE.IN & PIN5_bm) == PIN5_bm)
201     releaseDetEPS.RELEASE_DET_4 = 1;
202
203   // Enable sensor power
204   PORTK.OUTSET = (PIN2_bm | PIN3_bm); //V_SENSOR: EN10, PK2-3
205   enableEPS_9_10.ENABLE10 = 1;
206   PORTF.OUTSET = PIN6_bm; //nCS_sensor: PF6
207   #ifdef DEBUG
208     transmit_line((int8_t*) "\n\rEvent: V_SENSOR enabled");
209   #endif
210
211   // Apply default power scheme
212   power_subsystems(1,1,1,1,ENABLE); // OBDH and COMM default on
213
214   // Configure and start sensors
215   delay_ms(500); //wait until power is enabled. RC-circuit in enable signal
          need 477ms to charge
216   sensor_config();
217
218   // Initialize diagnostic values
219   diagnosticEPS.packageID = EPS_DIAGNOSTIC;
220   diagnosticEPS.epsID = EPS_I2C_ADDRESS;
221   diagnosticEPS.i2c_transaction_result = 0xFF; //default value
222   diagnosticEPS.i2c_bus_state = 0xFF; //default value
223
224   // Initialize statistics
225   clear_statistics();
226 }
227
228
229 void sensor_config(void){
230
231   uint8_t i;
232   uint8_t mode1;
233   uint8_t mode2;
234
235   #ifdef DEBUG
236     mode1 = VC_NORMAL;
237     mode2 = C_NORMAL;
238   #endif
239
240   #ifdef SENSOR
241     mode1 = VC_SENSOR;
242     mode2 = C_SENSOR;
243   #endif
244
245   #ifdef NORMAL
246     mode1 = VC_NORMAL;
247     mode2 = C_NORMAL;
248   #endif
249
250   // INA226 config
251   for(i=0; i<NR_OF_INA226_1; i++){
```

```
252
253     if(!ina226_config(ina226_i2c_adr_1[i], mode1, 0)){
254
255       gather_i2c_status();
256       internalFlagsEPS.i2c_transmission_error = 1;
257
258       #ifdef DEBUG
259         transmit_line((int8_t*) "\n\rError: config INA226 ");
260         transmit_hex(ina226_i2c_adr_1[i]);
261       #endif
262     }
263   }
264
265   for(i=0; i<NR_OF_INA226_2; i++){
266
267     if(!ina226_config(ina226_i2c_adr_2[i], mode2, 0)){
268
269       gather_i2c_status();
270       internalFlagsEPS.i2c_transmission_error = 1;
271
272       #ifdef DEBUG
273         transmit_line((int8_t*) "\n\rError: config INA226 ");
274         transmit_hex(ina226_i2c_adr_2[i]);
275       #endif
276     }
277   }
278
279   for(i=0; i<NR_OF_INA226_3; i++){
280
281     if(!ina226_config(ina226_i2c_adr_3[i], mode1, 0)){
282
283       gather_i2c_status();
284       internalFlagsEPS.i2c_transmission_error = 1;
285
286       #ifdef DEBUG
287         transmit_line((int8_t*) "\n\rError: config INA226 ");
288         transmit_hex(ina226_i2c_adr_3[i]);
289       #endif
290     }
291   }
292
293   // STC3100 config
294   for(i=0; i<NR_OF_STC3100; i++){
295
296     if(!stc3100_config(STC3100)){
297
298       gather_i2c_status();
299       internalFlagsEPS.i2c_transmission_error = 1;
300
301       #ifdef DEBUG
302         transmit_line((int8_t*) "\n\rError: config STC3100\t");
303       #endif
304     }
```

```
305   }
306 }
```

Listing E.4: EPS_cmd.c

```
1
2 /*
3  *  CubeSTAR EPS
4  *  Filename: EPS_cmd.c
5  *  Author: Knut Olav Skyttemyr
6  *  Description: EPS command handler
7  */
8
9 // Includes
10 #include "EPS.h"
11
12 // Local prototypes
13 uint32_t getAppCRC(void);
14 void restartEPS(void);
15 uint8_t normalValue(uint8_t *value, uint8_t nr);
16
17
18 void TWI_SlaveProcessData(void){
19
20   uint32_t crc_value;
21
22   switch(twiSlave.receivedData[0]){
23
24     case EPS_STATUS:
25       twiSlave.sendData[0] = HD4Value(&statusEPS.EPS_OFF);
26       commandsEPS.statusCMD = 1;
27       break;
28
29     case EPS_DIAGNOSTIC:
30       //Update diagnostic struct
31       diagnosticEPS.sysEventFlag = HD4Value(&sysEventEPS.SENSOR_I2C_ERROR);
32       diagnosticEPS.restartFlag = HD4Value(&resetEPS.rst_software);
33       diagnosticEPS.sysOnline = HD4Value(&systemsOnline.OBDH_ON);
34       diagnosticEPS.enable_1_4 = HD4Value(&enableEPS.ENABLE1);
35       diagnosticEPS.enable_5_8 = HD4Value(&enableEPS.ENABLE5);
36       diagnosticEPS.fault_1_4 = HD4Value(&faultEPS.FAULT1);
37       diagnosticEPS.enable9_10 = HD4Value(&enableEPS_9_10.ENABLE9);
38       diagnosticEPS.sensor_fault_1 = normalValue(&sensorFault1EPS.ina226_chg1
             , 8);
39       diagnosticEPS.sensor_fault_2 = normalValue(&sensorFault2EPS.ina226_obdh
             , 8);
40       diagnosticEPS.releaseStatus = HD4Value(&releaseDetEPS.RELEASE_DET_1);
41       diagnosticEPS.BW_set = BWSettingsEPS.BW_set.BW_set_16b;
42       diagnosticEPS.BW_clear = BWSettingsEPS.BW_clear.BW_clear_16b;
43       diagnosticEPS.TCS_state = TCSSettingsEPS.TCS_state;
44       diagnosticEPS.TCS_enable = TCSSettingsEPS.TCS_enable;
45       diagnosticEPS.TCS_disable = TCSSettingsEPS.TCS_disable;
46       diagnosticEPS.uptime_ticks = uptimeEPS.ticks.ticks_16b;
47       diagnosticEPS.uptime_weeks = uptimeEPS.weeks;
```

```
48        //Response is handled directly in I2C driver
49        commandsEPS.diagnosticCMD = 1;
50        break;
51
52      case EPS_APP_CRC:
53        crc_value = getAppCRC();
54        twiSlave.sendData[0] = (uint8_t) crc_value;
55        twiSlave.sendData[1] = (uint8_t) (crc_value >> 8);
56        twiSlave.sendData[2] = (uint8_t) (crc_value >> 16);
57        break;
58
59      case EPS_RESTART:
60        restartEPS();
61        break;
62
63      case EPS_BW_SETTINGS:
64        commandsEPS.BW_settings = 1;
65        BWSettingsEPS.BW_set.BW_set_lsb = twiSlave.receivedData[1];
66        BWSettingsEPS.BW_set.BW_set_msb = twiSlave.receivedData[2];
67        BWSettingsEPS.BW_clear.BW_clear_lsb = twiSlave.receivedData[3];
68        BWSettingsEPS.BW_clear.BW_clear_msb = twiSlave.receivedData[4];
69        break;
70
71      case EPS_TCS_SETTINGS:
72        commandsEPS.TCS_settings = 1;
73        TCSSettingsEPS.TCS_state = twiSlave.receivedData[1];
74        TCSSettingsEPS.TCS_enable = twiSlave.receivedData[2];
75        TCSSettingsEPS.TCS_disable = twiSlave.receivedData[3];
76        break;
77
78      case EPS_PWR_SUB:
79        //Update command list
80        commandsEPS.power = 1;
81        commandsEPS.powerMode = twiSlave.receivedData[1];
82        commandsEPS.powerSystem = twiSlave.receivedData[2];
83        twiSlave.sendData[0] = HD4Value(&systemsOnline.OBDH_ON); //return
              systems online flags
84        break;
85
86      case EPS_RELEASE:
87        //Update command list
88        commandsEPS.release = 1;
89        commandsEPS.releaseCMD = twiSlave.receivedData[1];
90        twiSlave.sendData[0] = HD4Value(&releaseDetEPS.RELEASE_DET_1); //return
               release status flag
91        break;
92
93      case EPS_UPTIME:
94        twiSlave.sendData[0] = uptimeEPS.ticks.ticks_msb;
95        twiSlave.sendData[1] = uptimeEPS.ticks.ticks_lsb;
96        twiSlave.sendData[2] = uptimeEPS.weeks;
97        break;
98
```

```
99      case EPS_CC_RST:
100       commandsEPS.CC_reset = 1;
101       break;
102
103     default:
104       twiSlave.abort = true; //NACK
105       statusEPS.EPS_USE = 1; //unhandled system error
106       sysEventEPS.UNKNOWN_CMD = 1;
107       #ifdef DEBUG
108         transmit_line((int8_t*) "\n\rEvent: Unknown system command");
109       #endif
110       break;
111   }
112 }
113
114
115 uint32_t getAppCRC(void){
116
117   uint32_t crc_value;
118   crc_value = SP_ApplicationCRC(); // crc check application
119   SP_WaitForSPM();
120
121   return crc_value;
122 }
123
124
125 void restartEPS(void){
126
127   ccp_write_io(&RST.CTRL, RST_SWRST_bm); //restart
128   delay_ms(1);
129 }
130
131
132 uint8_t HD4Value(register8_t *value){
133
134   uint8_t index = 0;
135
136   if(*value)
137     index += 1;
138   if(*(value+1))
139     index += 2;
140   if(*(value+2))
141     index += 4;
142   if(*(value+3))
143     index += 8;
144
145   return hamming_array[index];
146 }
147
148
149 bool HD4Valid(uint8_t HD4)
150 {
151    uint8_t bitcnt = 0;
```

```
152    uint8_t high, low;
153    high = HD4 & 0xF0; //Ex 20 = 20
154
155    if(high & 0x10) bitcnt++;
156    if(high & 0x20) bitcnt++; //Ex 20 -> bitcnt = 1;
157    if(high & 0x40) bitcnt++;
158    if(high & 0x80) bitcnt++;
159
160    if(bitcnt == 1 || bitcnt == 3) low = (~high>>4) & 0x0F; //Ex 20 and bitcnt
           is 1 -> low = DF>>4 = 0D
161    else low = (high>>4) & 0x0F; //Ex 30 = 03
162
163    return ((high|low) == HD4);
164 }
165
166
167 uint8_t normalValue(uint8_t *value, uint8_t nr){
168
169   uint8_t result = 0;
170   uint8_t i;
171
172   for(i=0; i<nr; i++){
173
174     if(*(value+i))
175       result |= (1<<i); //Merge flags into a byte
176   }
177
178   return result;
179 }
```

Listing E.5: housekeeping.c

```
1
2  /*
3   *  CubeSTAR EPS
4   *  Filename: housekeeping.c
5   *  Author: Knut Olav Skyttemyr
6   *  Description: Housekeeping functions, fetch and process sensor data
7   */
8
9  // Includes
10 #include "EPS.h"
11
12 // Local prototypes
13 void process_ina226(uint8_t *data, uint8_t mode, uint8_t i2c_address);
14 void process_tmp175(uint8_t *data, uint8_t i2c_address);
15 void process_stc3100(uint8_t *data);
16 void toggle_sensors(void);
17
18
19 void start_conversion(void){
20
21   // Start TMP conversion
22   for(uint8_t i=0; i<NR_OF_TMP175; i++){
```

```
23
24      if(!tmp175_config(tmp175_i2c_adr[i])){
25
26        gather_i2c_status();
27        internalFlagsEPS.i2c_transmission_error = 1;
28
29        #ifdef DEBUG
30          transmit_line((int8_t*) "\n\rError: start TMP175 ");
31          transmit_hex(tmp175_i2c_adr[i]);
32          TAB;
33        #endif
34      }
35    }
36  }
37
38
39  void fetch_sensordata(void){
40
41    uint8_t i;
42    uint8_t readData[8]; //Store sensor data locally to avoid losing data if
          interrupted
43
44    // Fetch INA226 sensor data
45    for(i=0; i<NR_OF_INA226_1; i++){
46
47      if(!ina226_fetch(ina226_i2c_adr_1[i], VOLTAGE)){
48
49        gather_i2c_status();
50
51        #ifdef DEBUG
52          transmit_line((int8_t*) "\n\rError: I2C INA226 voltage ");
53          transmit_hex(ina226_i2c_adr_1[i]);
54          TAB;
55        #endif
56
57        readData[0] = 0x80; //Set to non valid value: max negative value
58        readData[1] = 0x00;
59      }
60      else{
61        readData[0] = twiMaster.readData[0];
62        readData[1] = twiMaster.readData[1];
63      }
64
65      if(!ina226_fetch(ina226_i2c_adr_1[i], CURRENT)){
66
67        gather_i2c_status();
68
69        #ifdef DEBUG
70          transmit_line((int8_t*) "\n\rError: I2C INA226 current ");
71          transmit_hex(ina226_i2c_adr_1[i]);
72          TAB;
73        #endif
74
```

```
75        readData[2] = 0x80; //Set to non valid value: max negative value
76        readData[3] = 0x00;
77      }
78      else{
79        readData[2] = twiMaster.readData[0];
80        readData[3] = twiMaster.readData[1];
81      }
82
83      process_ina226(readData, V_AND_C, ina226_i2c_adr_1[i]);
84    }
85
86    // Fetch STC3100 data
87    for(i=0; i<NR_OF_STC3100; i++){
88
89      if(!stc3100_fetch(STC3100)){
90
91        gather_i2c_status();
92
93        #ifdef DEBUG
94          transmit_line((int8_t*) "\n\rError: I2C STC3100\t");
95        #endif
96
97        readData[0] = 0x00; //Set to non valid value: max negative value
98        readData[1] = 0x80;
99      }
100     else{
101
102       readData[0] = twiMaster.readData[0];
103       readData[1] = twiMaster.readData[1];
104     }
105
106     process_stc3100(readData);
107   }
108
109   // Fetch INA226 sensor data
110   for(i=0; i<NR_OF_INA226_2; i++){
111
112     if(!ina226_fetch(ina226_i2c_adr_2[i], CURRENT)){
113
114       gather_i2c_status();
115
116       #ifdef DEBUG
117         transmit_line((int8_t*) "\n\rError: I2C INA226 current ");
118         transmit_hex(ina226_i2c_adr_2[i]);
119         TAB;
120       #endif
121
122       readData[0] = 0x80; //Set to non valid value: max negative value
123       readData[1] = 0x00;
124     }
125     else{
126
127       readData[0] = twiMaster.readData[0];
```

```
128       readData[1] = twiMaster.readData[1];
129     }
130
131     process_ina226(readData, CURRENT, ina226_i2c_adr_2[i]);
132   }
133
134   // Fetch INA226 sensor data
135   for(i=0; i<NR_OF_INA226_3; i++){
136
137     if(!ina226_fetch(ina226_i2c_adr_3[i], VOLTAGE)){
138
139       gather_i2c_status();
140
141       #ifdef DEBUG
142         transmit_line((int8_t*) "\n\rError: I2C INA226 voltage ");
143         transmit_hex(ina226_i2c_adr_3[i]);
144         TAB;
145       #endif
146
147       readData[0] = 0x80; //Set to non valid value: max negative value
148       readData[1] = 0x00;
149     }
150     else{
151       readData[0] = twiMaster.readData[0];
152       readData[1] = twiMaster.readData[1];
153     }
154
155     if(!ina226_fetch(ina226_i2c_adr_3[i], CURRENT)){
156
157       gather_i2c_status();
158
159       #ifdef DEBUG
160         transmit_line((int8_t*) "\n\rError: I2C INA226 current ");
161         transmit_hex(ina226_i2c_adr_3[i]);
162         TAB;
163       #endif
164
165       readData[2] = 0x80; //Set to non valid value: max negative value
166       readData[3] = 0x00;
167     }
168     else{
169       readData[2] = twiMaster.readData[0];
170       readData[3] = twiMaster.readData[1];
171     }
172
173     process_ina226(readData, V_AND_C, ina226_i2c_adr_3[i]);
174   }
175
176   // Fetch TMP175 sensor data
177   for(i=0; i<NR_OF_TMP175; i++){
178
179     if(!tmp175_fetch(tmp175_i2c_adr[i])){
180
```

```
181       gather_i2c_status();
182
183       #ifdef DEBUG
184         transmit_line((int8_t*) "\n\rError: I2C TMP175 ");
185         transmit_hex(tmp175_i2c_adr[i]);
186         TAB;
187       #endif
188
189       readData[0] = 0x80; //Set to non valid value: max negative value
190       readData[1] = 0x00;
191     }
192     else{
193
194       readData[0] = twiMaster.readData[0];
195       readData[1] = twiMaster.readData[1];
196     }
197
198     process_tmp175(readData, tmp175_i2c_adr[i]);
199   }
200
201   error_handling();
202 }
203
204
205 void process_ina226(uint8_t *data, uint8_t mode, uint8_t i2c_address){
206
207   int16_t data_combined_1 = (data[0] << 8) | data[1]; //Combine MSB and LSB
        to 16bit signed
208   int16_t data_combined_2 = (data[2] << 8) | data[3];
209
210   #ifdef SENSOR
211
212     int8_t tx_buf[7]; //5 digits for 16 bit plus 2 digits for - and \0
213
214     if(mode == CURRENT){
215
216       itoa(data_combined_1 * 0.1, (char*) tx_buf, 10); //0.1mA/LSB
217       transmit_line(tx_buf);
218       TAB;
219     }
220     else if (mode == VOLTAGE){
221
222       itoa(data_combined_1 * 1.25, (char*) tx_buf, 10); //1.25mV/LSB
223       transmit_line(tx_buf);
224       TAB;
225     }
226     else if(mode == V_AND_C){
227
228       itoa(data_combined_1 * 1.25, (char*) tx_buf, 10); //1.25mV/LSB
229       transmit_line(tx_buf);
230       TAB;
231       itoa(data_combined_2 * 0.1, (char*) tx_buf, 10); //0.1mA/LSB
232       transmit_line(tx_buf);
```

```
233       TAB;
234     }
235
236   #endif
237
238   if(i2c_address == INA226_CHG1){
239
240     diagnosticEPS.vBat1 = data_combined_1;
241     diagnosticEPS.cCHG1 = data_combined_2;
242
243     // Voltage
244     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
245
246       sensorFault1EPS.ina226_chg1 = 1;
247
248       if(data_combined_1 == (int16_t) 0x8000){
249         internalFlagsEPS.i2c_transmission_error = 1;
250         sysEventEPS.SENSOR_I2C_ERROR = 1;
251       }
252       else{
253         internalFlagsEPS.sensor_value_invalid = 1;
254         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
255       }
256     }
257
258     else{
259       // Update statistics
260       if(data_combined_1 > diagnosticEPS.vBat1_max)
261         diagnosticEPS.vBat1_max = data_combined_1;
262       if(data_combined_1 < diagnosticEPS.vBat1_min)
263         diagnosticEPS.vBat1_min = data_combined_1;
264
265       // Battery at critical voltage level?
266       if((statusEPS.EPS_BW == 0) && (data_combined_1 < BWSettingsEPS.BW_set.
             BW_set_16b))
267         internalFlagsEPS.bw_alert++;
268
269       // Battery at safe voltage level?
270       if((statusEPS.EPS_BW == 1) && (data_combined_1 >= BWSettingsEPS.
             BW_clear.BW_clear_16b))
271         internalFlagsEPS.bw_clear++;
272     }
273
274     // Current
275     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
276
277       sensorFault1EPS.ina226_chg1 = 1;
278       internalFlagsEPS.sensor_value_invalid = 1;
279       sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
280     }
281
282     else{
283       // Update statistics
```

```
284       if(data_combined_2 > diagnosticEPS.cCHG1_max)
285         diagnosticEPS.cCHG1_max = data_combined_2;
286       if(data_combined_2 < diagnosticEPS.cCHG1_min)
287         diagnosticEPS.cCHG1_min = data_combined_2;
288     }
289   }
290
291   else if(i2c_address == INA226_CHG2){
292
293     diagnosticEPS.vBat2 = data_combined_1;
294     diagnosticEPS.cCHG2 = data_combined_2;
295
296     // Voltage
297     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
298
299       sensorFault1EPS.ina226_chg2 = 1;
300
301       if(data_combined_1 == (int16_t) 0x8000){
302         internalFlagsEPS.i2c_transmission_error = 1;
303         sysEventEPS.SENSOR_I2C_ERROR = 1;
304       }
305       else{
306         internalFlagsEPS.sensor_value_invalid = 1;
307         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
308       }
309     }
310
311     else{
312       // Update statistics
313       if(data_combined_1 > diagnosticEPS.vBat2_max)
314         diagnosticEPS.vBat2_max = data_combined_1;
315       if(data_combined_1 < diagnosticEPS.vBat2_min)
316         diagnosticEPS.vBat2_min = data_combined_1;
317
318       // Battery at critical voltage level?
319       if((statusEPS.EPS_BW == 0) && (data_combined_1 < BWSettingsEPS.BW_set.
             BW_set_16b))
320         internalFlagsEPS.bw_alert++;
321
322       // Battery at safe voltage level?
323       if((statusEPS.EPS_BW == 1) && (data_combined_1 >= BWSettingsEPS.
             BW_clear.BW_clear_16b))
324         internalFlagsEPS.bw_clear++;
325     }
326
327     // Current
328     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
329
330       sensorFault1EPS.ina226_chg2 = 1;
331       internalFlagsEPS.sensor_value_invalid = 1;
332       sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
333     }
334
```

```
335     else{
336       // Update statistics
337       if(data_combined_2 > diagnosticEPS.cCHG2_max)
338         diagnosticEPS.cCHG2_max = data_combined_2;
339       if(data_combined_2 < diagnosticEPS.cCHG2_min)
340         diagnosticEPS.cCHG2_min = data_combined_2;
341     }
342   }
343
344   else if(i2c_address == INA226_CHG3){
345
346     diagnosticEPS.vBat3 = data_combined_1;
347     diagnosticEPS.cCHG3 = data_combined_2;
348
349     // Voltage
350     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
351
352       sensorFault1EPS.ina226_chg3 = 1;
353
354       if(data_combined_1 == (int16_t) 0x8000){
355         internalFlagsEPS.i2c_transmission_error = 1;
356         sysEventEPS.SENSOR_I2C_ERROR = 1;
357       }
358       else{
359         internalFlagsEPS.sensor_value_invalid = 1;
360         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
361       }
362     }
363
364     else{
365       // Update statistics
366       if(data_combined_1 > diagnosticEPS.vBat3_max)
367         diagnosticEPS.vBat3_max = data_combined_1;
368       if(data_combined_1 < diagnosticEPS.vBat3_min)
369         diagnosticEPS.vBat3_min = data_combined_1;
370
371       // Battery at critical voltage level?
372       if((statusEPS.EPS_BW == 0) && (data_combined_1 < BWSettingsEPS.BW_set.
             BW_set_16b))
373         internalFlagsEPS.bw_alert++;
374
375       // Battery at safe voltage level?
376       if((statusEPS.EPS_BW == 1) && (data_combined_1 >= BWSettingsEPS.
             BW_clear.BW_clear_16b))
377         internalFlagsEPS.bw_clear++;
378     }
379
380     // Current
381     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
382
383       sensorFault1EPS.ina226_chg3 = 1;
384       internalFlagsEPS.sensor_value_invalid = 1;
385       sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
```

```
386     }
387
388     else{
389       // Update statistics
390       if(data_combined_2 > diagnosticEPS.cCHG3_max)
391         diagnosticEPS.cCHG3_max = data_combined_2;
392       if(data_combined_2 < diagnosticEPS.cCHG3_min)
393         diagnosticEPS.cCHG3_min = data_combined_2;
394     }
395   }
396
397   else if(i2c_address == INA226_CHG4){
398
399     diagnosticEPS.vBat4 = data_combined_1;
400     diagnosticEPS.cCHG4 = data_combined_2;
401
402     // Voltage
403     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
404
405       sensorFault1EPS.ina226_chg4 = 1;
406
407       if(data_combined_1 == (int16_t) 0x8000){
408         internalFlagsEPS.i2c_transmission_error = 1;
409         sysEventEPS.SENSOR_I2C_ERROR = 1;
410       }
411       else{
412         internalFlagsEPS.sensor_value_invalid = 1;
413         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
414       }
415     }
416
417     else{
418       // Update statistics
419       if(data_combined_1 > diagnosticEPS.vBat4_max)
420         diagnosticEPS.vBat4_max = data_combined_1;
421       if(data_combined_1 < diagnosticEPS.vBat4_min)
422         diagnosticEPS.vBat4_min = data_combined_1;
423
424       // Battery at critical voltage level?
425       if((statusEPS.EPS_BW == 0) && (data_combined_1 < BWSettingsEPS.BW_set.
             BW_set_16b))
426         internalFlagsEPS.bw_alert++;
427
428       // Battery at safe voltage level?
429       if((statusEPS.EPS_BW == 1) && (data_combined_1 >= BWSettingsEPS.
             BW_clear.BW_clear_16b))
430         internalFlagsEPS.bw_clear++;
431     }
432
433     // Current
434     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
435
436       sensorFault1EPS.ina226_chg4 = 1;
```

```
437       internalFlagsEPS.sensor_value_invalid = 1;
438       sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
439     }
440
441     else{
442       // Update statistics
443       if(data_combined_2 > diagnosticEPS.cCHG4_max)
444         diagnosticEPS.cCHG4_max = data_combined_2;
445       if(data_combined_2 < diagnosticEPS.cCHG4_min)
446         diagnosticEPS.cCHG4_min = data_combined_2;
447     }
448   }
449
450   else if(i2c_address == INA226_BAT){
451
452     diagnosticEPS.cMainBus = data_combined_1;
453
454     if ((data_combined_1 < CURRENT_MIN) || (data_combined_1 > CURRENT_MAX)){
455
456       sensorFault1EPS.ina226_mainbus = 1;
457
458       if(data_combined_1 == (int16_t) 0x8000){
459         internalFlagsEPS.i2c_transmission_error = 1;
460         sysEventEPS.SENSOR_I2C_ERROR = 1;
461       }
462       else{
463         internalFlagsEPS.sensor_value_invalid = 1;
464         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
465       }
466     }
467
468     else{
469       // Update statistics
470       if(data_combined_1 > diagnosticEPS.cMainBus_max)
471         diagnosticEPS.cMainBus_max = data_combined_1;
472       if(data_combined_1 < diagnosticEPS.cMainBus_min)
473         diagnosticEPS.cMainBus_min = data_combined_1;
474     }
475   }
476
477   else if(i2c_address == INA226_MAINBUS){
478
479     //do nothing. Sensor not in use.
480   }
481
482   else if(i2c_address == INA226_PLD){
483
484     diagnosticEPS.vPLD = data_combined_1;
485     diagnosticEPS.cPLD = data_combined_2;
486
487     // Voltage
488     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
489
```

```
490       sensorFault1EPS.ina226_payload = 1;
491
492       if(data_combined_1 == (int16_t) 0x8000){
493         internalFlagsEPS.i2c_transmission_error = 1;
494         sysEventEPS.SENSOR_I2C_ERROR = 1;
495       }
496       else{
497         internalFlagsEPS.sensor_value_invalid = 1;
498         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
499       }
500     }
501
502     else{
503       // Update statistics
504       if(data_combined_1 > diagnosticEPS.vPLD_max)
505         diagnosticEPS.vPLD_max = data_combined_1;
506       if(data_combined_1 < diagnosticEPS.vPLD_min)
507         diagnosticEPS.vPLD_min = data_combined_1;
508     }
509
510     // Current
511     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
512
513       sensorFault1EPS.ina226_payload = 1;
514
515       if(data_combined_2 == (int16_t) 0x8000){
516         internalFlagsEPS.i2c_transmission_error = 1;
517         sysEventEPS.SENSOR_I2C_ERROR = 1;
518       }
519       else{
520         internalFlagsEPS.sensor_value_invalid = 1;
521         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
522       }
523     }
524
525     else{
526       // Update statistics
527       if(data_combined_2 > diagnosticEPS.cPLD_max)
528         diagnosticEPS.cPLD_max = data_combined_2;
529       if(data_combined_2 < diagnosticEPS.cPLD_min)
530         diagnosticEPS.cPLD_min = data_combined_2;
531     }
532   }
533
534   else if(i2c_address == INA226_ADCS){
535
536     diagnosticEPS.vADCS = data_combined_1;
537     diagnosticEPS.cADCS = data_combined_2;
538
539     // Voltage
540     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
541
542       sensorFault1EPS.ina226_adcs = 1;
```

```
543
544       if(data_combined_1 == (int16_t) 0x8000){
545         internalFlagsEPS.i2c_transmission_error = 1;
546         sysEventEPS.SENSOR_I2C_ERROR = 1;
547       }
548       else{
549         internalFlagsEPS.sensor_value_invalid = 1;
550         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
551       }
552     }
553
554     else{
555
556       // Update statistics
557       if(data_combined_1 > diagnosticEPS.vADCS_max)
558         diagnosticEPS.vADCS_max = data_combined_1;
559       if(data_combined_1 < diagnosticEPS.vADCS_min)
560         diagnosticEPS.vADCS_min = data_combined_1;
561     }
562
563     // Current
564     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
565
566       sensorFault1EPS.ina226_adcs = 1;
567
568       if(data_combined_2 == (int16_t) 0x8000){
569         internalFlagsEPS.i2c_transmission_error = 1;
570         sysEventEPS.SENSOR_I2C_ERROR = 1;
571       }
572       else{
573         internalFlagsEPS.sensor_value_invalid = 1;
574         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
575       }
576     }
577
578     else{
579       // Update statistics
580       if(data_combined_2 > diagnosticEPS.cADCS_max)
581         diagnosticEPS.cADCS_max = data_combined_2;
582       if(data_combined_2 < diagnosticEPS.cADCS_min)
583         diagnosticEPS.cADCS_min = data_combined_2;
584     }
585   }
586
587   else if(i2c_address == INA226_OBDH){
588
589     diagnosticEPS.vOBDH = data_combined_1;
590     diagnosticEPS.cOBDH = data_combined_2;
591
592     // Voltage
593     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
594
595       sensorFault2EPS.ina226_obdh = 1;
```

```
596
597       if(data_combined_1 == (int16_t) 0x8000){
598         internalFlagsEPS.i2c_transmission_error = 1;
599         sysEventEPS.SENSOR_I2C_ERROR = 1;
600       }
601       else{
602         internalFlagsEPS.sensor_value_invalid = 1;
603         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
604       }
605     }
606
607     else{
608       // Update statistics
609       if(data_combined_1 > diagnosticEPS.vOBDH_max)
610         diagnosticEPS.vOBDH_max = data_combined_1;
611       if(data_combined_1 < diagnosticEPS.vOBDH_min)
612         diagnosticEPS.vOBDH_min = data_combined_1;
613     }
614
615     // Current
616     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
617
618       sensorFault2EPS.ina226_obdh = 1;
619
620       if(data_combined_2 == (int16_t) 0x8000){
621         internalFlagsEPS.i2c_transmission_error = 1;
622         sysEventEPS.SENSOR_I2C_ERROR = 1;
623       }
624       else{
625         internalFlagsEPS.sensor_value_invalid = 1;
626         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
627       }
628     }
629
630     else{
631       // Update statistics
632       if(data_combined_2 > diagnosticEPS.cOBDH_max)
633         diagnosticEPS.cOBDH_max = data_combined_2;
634       if(data_combined_2 < diagnosticEPS.cOBDH_min)
635         diagnosticEPS.cOBDH_min = data_combined_2;
636     }
637   }
638
639   else if(i2c_address == INA226_COMM){
640
641     diagnosticEPS.vCOMM = data_combined_1;
642     diagnosticEPS.cCOMM = data_combined_2;
643
644     // Voltage
645     if((data_combined_1 < V_PWR_MIN) || (data_combined_1 > V_PWR_MAX)){
646
647       sensorFault2EPS.ina226_comm = 1;
648
```

```
649       if(data_combined_1 == (int16_t) 0x8000){
650         internalFlagsEPS.i2c_transmission_error = 1;
651         sysEventEPS.SENSOR_I2C_ERROR = 1;
652       }
653       else{
654         internalFlagsEPS.sensor_value_invalid = 1;
655         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
656       }
657     }
658
659     else{
660       // Update statistics
661       if(data_combined_1 > diagnosticEPS.vCOMM_max)
662         diagnosticEPS.vCOMM_max = data_combined_1;
663       if(data_combined_1 < diagnosticEPS.vCOMM_min)
664         diagnosticEPS.vCOMM_min = data_combined_1;
665
666       // Check if power distribution to COMM is working, if not toggle switch
            .
667       if(comm_switch == 0){
668
669         if((data_combined_1 < VCOMM_MIN) && (enableEPS.ENABLE4 == 1) && (
              faultEPS.FAULT4 == 0)){
670           comm_switch ^= 1;
671           power_subsystems(0,0,1,0, ENABLE);
672           statusEPS.EPS_NSE = 1; //New system event
673         }
674       }
675       else if(comm_switch == 1){
676
677         if((data_combined_1 < VCOMM_MIN) && (enableEPS.ENABLE5 == 1) && (
              enableEPS_9_10.FAULT5 == 0)){
678           comm_switch ^= 1;
679           power_subsystems(0,0,1,0, ENABLE);
680           statusEPS.EPS_NSE = 1; //New system event
681         }
682       }
683       else{
684
685         statusEPS.EPS_USE = 1; //unhandled system error
686         sysEventEPS.UNKNOWN_STATE = 1;
687         #ifdef DEBUG
688           transmit_line((int8_t*)"\n\rError: Unknown state, comm_switch flag"
                );
689         #endif
690       }
691     }
692
693     // Current
694     if ((data_combined_2 < CURRENT_MIN) || (data_combined_2 > CURRENT_MAX)){
695
696       sensorFault2EPS.ina226_comm = 1;
697
```

```
698       if(data_combined_2 == (int16_t) 0x8000){
699         internalFlagsEPS.i2c_transmission_error = 1;
700         sysEventEPS.SENSOR_I2C_ERROR = 1;
701       }
702       else{
703         internalFlagsEPS.sensor_value_invalid = 1;
704         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
705       }
706     }
707
708     else{
709       // Update statistics
710       if(data_combined_2 > diagnosticEPS.cCOMM_max)
711         diagnosticEPS.cCOMM_max = data_combined_2;
712       if(data_combined_2 < diagnosticEPS.cCOMM_min)
713         diagnosticEPS.cCOMM_min = data_combined_2;
714     }
715   }
716
717   else{
718
719     sysEventEPS.UNKNOWN_STATE = 1;
720     statusEPS.EPS_USE = 1;
721     #ifdef DEBUG
722       transmit_line((int8_t*)"\n\rError: Unknown state process_ina226()\t");
723     #endif
724   }
725
726   // Set battery warning flag if voltage is below a certain level
727   if((internalFlagsEPS.bw_alert >= 2) && (statusEPS.EPS_BW == 0)){
728
729     statusEPS.EPS_BW = 1;
730     PORTB.OUTSET = PIN2_bm; //set EPS_IRQ high to alert the OBDH
731     #ifdef DEBUG
732       transmit_line((int8_t*)"\n\rEvent: Battery under-voltage warning\n\r");
733     #endif
734   }
735
736   // Clear battery warning flag if voltage is above a certain level
737   if((internalFlagsEPS.bw_clear >= 2) && (statusEPS.EPS_BW == 1)){
738
739     statusEPS.EPS_BW = 0;
740     PORTB.OUTCLR = PIN2_bm; //Clear EPS_IRQ
741     #ifdef DEBUG
742       transmit_line((int8_t*)"\n\rEvent: Battery warning cleared\n\r");
743     #endif
744   }
745 }
746
747
748 void process_tmp175(uint8_t *data, uint8_t i2c_address){
749
750   int8_t data_combined = data[0]; //Ignore data[1] -> 8-bit resolution
```

```
751
752   #ifdef SENSOR
753
754     int8_t tx_buf[7]; //5 digits for 16 bit plus 2 digits for - and \0
755     itoa((data_combined), (char*) tx_buf, 10);
756     transmit_line(tx_buf);
757     TAB;
758
759   #endif
760
761   // Store sensor data in diagnostic struct
762   if(i2c_address == TMP175_BACKPANEL){
763
764     diagnosticEPS.tmpBackPanel = data_combined;
765
766     if((data_combined < TEMP_MIN) || (data_combined > TEMP_MAX)){
767
768       // Do not update statistics because of value out of range
769
770       // Update flags
771       sensorFault2EPS.tmp175_backpanel = 1;
772
773       if(data_combined == (int8_t) 0x80){
774         internalFlagsEPS.i2c_transmission_error = 1;
775         sysEventEPS.SENSOR_I2C_ERROR = 1;
776       }
777       else{
778         internalFlagsEPS.sensor_value_invalid = 1;
779         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
780       }
781     }
782
783     else{
784       // Update statistics
785       if(data_combined > diagnosticEPS.tmpBackPanel_max)
786         diagnosticEPS.tmpBackPanel_max = data_combined;
787       if(data_combined < diagnosticEPS.tmpBackPanel_min)
788         diagnosticEPS.tmpBackPanel_min = data_combined;
789     }
790   }
791
792   else if(i2c_address == TMP175_BAT1){
793
794     diagnosticEPS.tmpBat1 = data_combined;
795
796     if((data_combined < TEMP_MIN) || (data_combined > TEMP_MAX)){
797
798       // Do not update statistics because of value out of range
799
800       // Update flags
801       sensorFault2EPS.tmp175_bat1 = 1;
802
803       if(data_combined == (int8_t) 0x80){
```

```
804         internalFlagsEPS.i2c_transmission_error = 1;
805         sysEventEPS.SENSOR_I2C_ERROR = 1;
806       }
807       else{
808         internalFlagsEPS.sensor_value_invalid = 1;
809         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
810       }
811     }
812
813     else{
814       // Update statistics
815       if(data_combined > diagnosticEPS.tmpBat1_max)
816         diagnosticEPS.tmpBat1_max = data_combined;
817       if(data_combined < diagnosticEPS.tmpBat1_min)
818         diagnosticEPS.tmpBat1_min = data_combined;
819
820       // Check if TCS should be enabled or disabled
821       if((data_combined <= TCSSettingsEPS.TCS_enable) && (internalFlagsEPS.
              bat_under_temp == 0))
822         internalFlagsEPS.lowTemp++;
823       else if((data_combined >= TCSSettingsEPS.TCS_disable) && (
              internalFlagsEPS.bat_under_temp == 1))
824         internalFlagsEPS.normalTemp++;
825     }
826   }
827
828   else if(i2c_address == TMP175_BAT2){
829
830     diagnosticEPS.tmpBat2 = data_combined;
831
832     if((data_combined < TEMP_MIN) || (data_combined > TEMP_MAX)){
833
834       // Do not update statistics because of value out of range
835
836       // Update flags
837       sensorFault2EPS.tmp175_bat2 = 1;
838
839       if(data_combined == (int8_t) 0x80){
840         internalFlagsEPS.i2c_transmission_error = 1;
841         sysEventEPS.SENSOR_I2C_ERROR = 1;
842       }
843       else{
844         internalFlagsEPS.sensor_value_invalid = 1;
845         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
846       }
847     }
848
849     else{
850       // Update statistics
851       if(data_combined > diagnosticEPS.tmpBat2_max)
852         diagnosticEPS.tmpBat2_max = data_combined;
853       if(data_combined < diagnosticEPS.tmpBat2_min)
854         diagnosticEPS.tmpBat2_min = data_combined;
```

```
855
856       // Check if TCS should be enabled or disabled
857       if((data_combined <= TCSSettingsEPS.TCS_enable) && (internalFlagsEPS.
              bat_under_temp == 0))
858         internalFlagsEPS.lowTemp++;
859       else if((data_combined >= TCSSettingsEPS.TCS_disable) && (
              internalFlagsEPS.bat_under_temp == 1))
860         internalFlagsEPS.normalTemp++;
861     }
862   }
863
864   else if(i2c_address == TMP175_BAT3){
865
866     diagnosticEPS.tmpBat3 = data_combined;
867
868     if((data_combined < TEMP_MIN) || (data_combined > TEMP_MAX)){
869
870       // Do not update statistics because of value out of range
871
872       // Update flags
873       sensorFault2EPS.tmp175_bat3 = 1;
874
875       if(data_combined == (int8_t) 0x80){
876         internalFlagsEPS.i2c_transmission_error = 1;
877         sysEventEPS.SENSOR_I2C_ERROR = 1;
878       }
879       else{
880         internalFlagsEPS.sensor_value_invalid = 1;
881         sysEventEPS.SENSOR_OUT_OF_RANGE = 1;
882       }
883     }
884
885     else{
886       // Update statistics
887       if(data_combined > diagnosticEPS.tmpBat3_max)
888         diagnosticEPS.tmpBat3_max = data_combined;
889       if(data_combined < diagnosticEPS.tmpBat3_min)
890         diagnosticEPS.tmpBat3_min = data_combined;
891
892       // Check if TCS should be enabled or disabled
893       if((data_combined <= TCSSettingsEPS.TCS_enable) && (internalFlagsEPS.
              bat_under_temp == 0))
894         internalFlagsEPS.lowTemp++;
895       else if((data_combined >= TCSSettingsEPS.TCS_disable) && (
              internalFlagsEPS.bat_under_temp == 1))
896         internalFlagsEPS.normalTemp++;
897     }
898   }
899
900   else{
901
902     sysEventEPS.UNKNOWN_STATE = 1;
903     statusEPS.EPS_USE = 1;
```

```
904     #ifdef DEBUG
905       transmit_line((int8_t*)"\n\rError: Unknown state process_tmp175()\t");
906     #endif
907   }
908
909   if((internalFlagsEPS.bat_under_temp == 0) && (internalFlagsEPS.lowTemp >=
          2) && (statusEPS.EPS_BW == 0) && (TCSSettingsEPS.TCS_state == 1)){
910
911     // Enable TCS
912     PORTK.OUTSET = PIN4_bm | PIN5_bm;
913     enableEPS_9_10.TCS_EN = 1;
914     statusEPS.EPS_NSE = 1;
915     internalFlagsEPS.bat_under_temp = 1;
916
917     #ifdef DEBUG
918       transmit_line((int8_t*) "\n\rAlert: TCS enabled\t");
919     #endif
920   }
921   else if(((internalFlagsEPS.bat_under_temp == 1) && (internalFlagsEPS.
          normalTemp >= 2)) || ((TCSSettingsEPS.TCS_state == 0) && (
          internalFlagsEPS.TCS_off == 1)) || ((statusEPS.EPS_BW == 1) && (
          internalFlagsEPS.bat_under_temp == 1))){
922
923     // Disable TCS
924     PORTK.OUTCLR = PIN4_bm | PIN5_bm;
925     enableEPS_9_10.TCS_EN = 0;
926     statusEPS.EPS_NSE = 1;
927     internalFlagsEPS.bat_under_temp = 0;
928     internalFlagsEPS.TCS_off = 0;
929
930     #ifdef DEBUG
931       transmit_line((int8_t*) "\n\rAlert: TCS disabled\t");
932     #endif
933   }
934 }
935
936
937 void process_stc3100(uint8_t *data){
938
939   int16_t data_combined = (data[1] << 8) | data[0]; //Combine MSB and LSB to
          16bit. STC3100: LSB first
940
941   #ifdef SENSOR
942
943     int8_t tx_buf[7]; //5 digits for 16 bit plus 2 digits for - and \0
944
945     // Charge, 16 bit.
946     itoa(data_combined * 0.268, (char*) tx_buf, 10); //6.7uVh/25mohm=0.268
947     transmit_line(tx_buf);
948     TAB;
949
950   #endif
951
```

```
952   diagnosticEPS.batSoC = data_combined;
953
954   if(data_combined == (int16_t) 0x8000){
955     sensorFault1EPS.stc3100 = 1;
956     internalFlagsEPS.i2c_transmission_error = 1;
957     sysEventEPS.SENSOR_I2C_ERROR = 1;
958   }
959 }
960
961
962 void error_handling(void){
963
964   if((internalFlagsEPS.i2c_transmission_error == 1) || (internalFlagsEPS.
        sensor_value_invalid == 1)){
965
966     #ifdef DEBUG
967
968       if(internalFlagsEPS.i2c_transmission_error == 1){
969
970         transmit_line((int8_t*) "\n\rError type: I2C transmission error");
971         transmit_line((int8_t*) "\tI2C status: ");
972         transmit_hex(diagnosticEPS.i2c_transaction_result);
973         TAB;
974         transmit_hex(diagnosticEPS.i2c_bus_state);
975       }
976       if(internalFlagsEPS.sensor_value_invalid == 1)
977         transmit_line((int8_t*) "\n\rError type: Sensor value invalid");
978
979       transmit_line((int8_t*) "\n\rEvent: Sensor toggle");
980     #endif
981
982     toggle_sensors();
983     statusEPS.EPS_NSE = 1; //New system event
984
985     // Clear internal flags
986     internalFlagsEPS.i2c_transmission_error = 0;
987     internalFlagsEPS.sensor_value_invalid = 0;
988   }
989 }
990
991
992 void toggle_sensors(void){
993
994   PORTK.OUTCLR = (PIN2_bm | PIN3_bm); //V_SENSOR: EN10, PK2-3
995   enableEPS_9_10.ENABLE10 = 0;
996   PORTF.OUTCLR = PIN6_bm; //nCS_sensor: PF6
997   delay_ms(POWER_TOGGLE_DLY);
998   PORTK.OUTSET = (PIN2_bm | PIN3_bm);
999   enableEPS_9_10.ENABLE10 = 1;
1000  PORTF.OUTSET = PIN6_bm; //nCS_sensor: PF6
1001  delay_ms(10); //Wait until power is enabled
1002  sensor_config(); //Re-initialize sensors
1003 }
```

```

void gather_i2c_status(void){

  //log I2C status in case of error
  diagnosticEPS.i2c_transaction_result = twiMaster.result;
  diagnosticEPS.i2c_bus_state = I2C_BUS4.MASTER.STATUS;
}
```

Listing E.6: sensor_i2c_cmd.c

```

/*
 *  CubeSTAR EPS
 *  Filename: sensor_i2c_cmd.c
 *  Author: Knut Olav Skyttemyr
 *  Description: I2C commands for EPS sensors
 */

// Includes
#include "EPS.h"


void i2c_ready(void){

  uint8_t i = 0;

  for(i = 0; i < TIMEOUT_I2C; i++){
    if(twiMaster.result == TWIM_RESULT_OK)
      break;
    delay_us(100);
  }
}


uint8_t tmp175_config(uint8_t i2c_address){

  uint8_t sendBuffer[2];
  sendBuffer[0] = 0b01; //Configuration register
  sendBuffer[1] = 0b10000001; //9-bit resolution, one-shot
  TWI_MasterWrite(&twiMaster, i2c_address, sendBuffer, 2);
  i2c_ready();
  if(twiMaster.result != TWIM_RESULT_OK)
    return 0;
  else
    return 1;
}


uint8_t tmp175_fetch(uint8_t i2c_address){

  uint8_t sendBuffer = 0; //Temp register, MSB first
  TWI_MasterWriteRead(&twiMaster, i2c_address, &sendBuffer, 1, 2); //Write 1,
      read 2 bytes
```

```
43   i2c_ready();
44   if(twiMaster.result != TWIM_RESULT_OK)
45     return 0;
46   else
47     return 1;
48 }
49
50
51 uint8_t ina226_config(uint8_t i2c_address, uint8_t mode, uint8_t mask){
52
53   uint8_t error = 0;
54   uint8_t sendBuffer[3];
55
56   // Configuration
57   sendBuffer[0] = 0x00;
58   if(mode == VC_SENSOR){
59     sendBuffer[1] = 0b01001110; //number of avg: 1024, voltage conversion
         time: 588us
60     sendBuffer[2] = 0b11110111; //current conversion time: 4.156ms, mode:
         Shunt&bus continuous. Total 4.858 seconds
61   }
62   else if(mode == C_SENSOR){
63     sendBuffer[1] = 0b01001110; //number of avg: 1024, voltage conversion
         time: 588us
64     sendBuffer[2] = 0b11110101; //current conversion time: 4.156ms, mode:
         Shunt continuous. Total 4.858 seconds
65   }
66   else if(mode == VC_NORMAL){
67     sendBuffer[1] = 0b01001111; //number of avg: 1024, voltage conversion
         time: 1.1ms
68     sendBuffer[2] = 0b00111111; //current conversion time: 8.244ms, mode:
         Shunt&bus continuous. Total 9.568 seconds
69   }
70   else if(mode == C_NORMAL){
71     sendBuffer[1] = 0b01001111; //number of avg: 1024, voltage conversion
         time: 1.1ms
72     sendBuffer[2] = 0b00111101; //current conversion time: 8.244ms, mode:
         Shunt continuous. Total 9.568 seconds
73   }
74   else{
75
76     statusEPS.EPS_USE = 1; //unhandled system error
77     sysEventEPS.UNKNOWN_STATE = 1;
78     #ifdef DEBUG
79       transmit_line((int8_t*)"\n\rError: Unknown state, INA226 init");
80     #endif
81   }
82
83   TWI_MasterWrite(&twiMaster, i2c_address, sendBuffer, 3);
84   i2c_ready();
85   if(twiMaster.result != TWIM_RESULT_OK)
86     error = 1;
87
```

```
88    // Calibration
89    sendBuffer[0] = 0x05;
90    sendBuffer[1] = 0x08; //currentLSB = 3A/2^15 ~= 0.1mA -> CAL = 0.00512/(0.1
          mA*25mohm) = 0x0800.
91    sendBuffer[2] = 0x00;
92    TWI_MasterWrite(&twiMaster, i2c_address, sendBuffer, 3);
93    i2c_ready();
94    if(twiMaster.result != TWIM_RESULT_OK)
95      error = 1;
96
97    if(error == 1)
98      return 0;
99    else
100     return 1;
101 }
102
103
104 uint8_t ina226_fetch(uint8_t i2c_address, uint8_t mode){
105
106   uint8_t sendBuffer;
107
108   if(mode == CURRENT){
109     sendBuffer = 4; // 0x04 = Current register
110     TWI_MasterWriteRead(&twiMaster, i2c_address, &sendBuffer, 1, 2);
111     i2c_ready();
112     if(twiMaster.result != TWIM_RESULT_OK)
113       return 0;
114     else
115       return 1;
116   }
117   else if(mode == VOLTAGE){
118     sendBuffer = 2; // 0x02 = Voltage register
119     TWI_MasterWriteRead(&twiMaster, i2c_address, &sendBuffer, 1, 2);
120     i2c_ready();
121     if(twiMaster.result != TWIM_RESULT_OK)
122       return 0;
123     else
124       return 1;
125   }
126   else{
127
128     sysEventEPS.UNKNOWN_STATE = 1;
129     statusEPS.EPS_USE = 1;
130     #ifdef DEBUG
131       transmit_line((int8_t*)"\n\rError: Unknown state ina226_fetch()");
132     #endif
133     return 0;
134   }
135 }
136
137
138 uint8_t stc3100_config(uint8_t i2c_address){
139
```

```
140   uint8_t sendBuffer[2];
141   sendBuffer[0] = 0; // 0x00 = mode register
142   sendBuffer[1] = 0x10; //Operating mode, 14 bit
143   TWI_MasterWrite(&twiMaster, i2c_address, sendBuffer, 2);
144   i2c_ready();
145   if(twiMaster.result != TWIM_RESULT_OK)
146     return 0;
147   else
148     return 1;
149 }
150
151
152 uint8_t stc3100_fetch(uint8_t i2c_address){
153
154   uint8_t sendBuffer = 2; //register address 0x02: charge (LSB first)
155   TWI_MasterWriteRead(&twiMaster, i2c_address, &sendBuffer, 1, 2);
156   i2c_ready();
157   if(twiMaster.result != TWIM_RESULT_OK)
158     return 0;
159   else
160     return 1;
161 }
162
163
164 uint8_t stc3100_reset(uint8_t i2c_address){
165
166   uint8_t sendBuffer[2];
167   sendBuffer[0] = 1; // 0x01 = ctrl register
168   sendBuffer[1] = 0x02; //Reset
169   TWI_MasterWrite(&twiMaster, i2c_address, sendBuffer, 2);
170   i2c_ready();
171   if(twiMaster.result != TWIM_RESULT_OK)
172     return 0;
173   else
174     return 1;
175 }
```

Listing E.7: uart.c

```
1  /*
2   *  CubeSTAR EPS
3   *  Filename: uart.c
4   *  Author: Knut Olav Skyttemyr
5   *  Description: UART functions
6   */
7
8  // Includes
9  #include "EPS.h"
10
11
12 void transmit(int8_t data){
13
14   while((UART.STATUS & USART_DREIF_bm) == 0); //while buffer is full
15   UART.DATA = data;
```

```
16 }
17
18
19 void transmit_line(int8_t *line){
20
21   for(uint8_t i=0; line[i] != '\0'; i++){
22     transmit(line[i]);
23   }
24 }
25
26
27 void transmit_hex(int8_t data){
28
29   int8_t tx_buf[] = "0x  ";
30   int8_t temp1 = ((data & 0xF0)>>4) + 0x30;
31   if(temp1 > 0x39){
32     temp1 += 7;
33   }
34   int8_t temp2 = (data & 0x0F) + 0x30;
35   if(temp2 > 0x39){
36     temp2 += 7;
37   }
38   tx_buf[2] = temp1;
39   tx_buf[3] = temp2;
40   transmit_line(tx_buf);
41 }
```

Appendix F

CD-ROM

- PDF version of the master thesis
- Back Panel circuit schematics
- Battery module schematics and PCB layout
- SPV1040 test board and Labview program
- AVR studio project and source code
- Component list with ordering numbers
- EPS user manual. Document with more details about operation and suggested improvements of the EPS
- Sensor error calculation spreadsheet